National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

Canada

NL-339 (r. 86/08)

The University of Alberta

A Multiple-processor Architecture for Image Processing

© De-lei Lee

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta
Spring, 1987

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-37648-1

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: De-lei Lee

TITLE OF THESIS: A Multiple-processor Architecture for Image Processing

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1987

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Debi Lee

Permanent Address:
Department of Computer Science
York University
Downsview, Ontario
M3J 1P3 Canada

Dated 12 March 1987

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Multiple-processor Architecture for Image Processing** submitted by **De-lei Lee** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy.**

W. A. Davis
.....
Supervisor

Z. J. Koles
.....

.....

Xiaobo Li
.....

McLeish
.....

A. P. Reeves
.....
External Examiner

Date 10 Apr 87

Dedicated to mom and dad
with love and admiration

Abstract

This thesis is concerned with the design and analysis of a parallel computer architecture for image processing. The proposed architecture is capable of supporting at high efficiency a wide range of image processing tasks that are diverse in their computational structures by means of dynamic algorithm-architecture matching. This architectural versatility is achieved primarily by making use of a novel storage scheme in conjunction with a unique multistage interconnection network which allow: 1) various N-partitions of an image; 2) parallel access of any part of an N-partition of the image without memory conflict; 3) placement of the elements of a given part on a processing element array in useful orders for subsequent highly parallel processing. Fundamental properties of the architecture are given, and some implementation considerations are provided. The use of the architecture for the efficient parallel implementation of three fundamental classes of computationally intensive image processing tasks is investigated. They are local neighborhood operations, global neighborhood operations, and global transforms. It is shown that although they are diverse in their computational structures, the architecture allows them to achieve a high degree of parallelism in the computation.

Acknowledgements

I would like to take this opportunity to thank the many people who have graciously helped in making this work possible.

My first acknowledgement must go to my supervisor Dr. Wayne A. Davis for his advise, encouragement, and many exciting discussions throughout the course of this research. Wayne's patience and understanding have allowed my work to progress further than I could have expected. Further appreciations are due to the members of my examining committee, Drs. J. Tartar, Z.J. Koles, T.M. Caelli, and X. Li, for their time and helpful comments on this work. In particular, I would like to thank Dr. Anthony P. Reeves, my external examiner from Cornell University, who has been invaluable in his thorough assessment of this dissertation.

I wish to express my sincere gratitude to my close friends, Eric and Erika Schmitz, for their parental love and for being there whenever I needed them. All people in the computing science department have contributed to making my study here a very enjoyable experience.

Special thanks must go to my wife, Xiaoning, without whose love and support this thesis would never have been completed. Above all, I am greatly indebted to my parents, Ruqi Lee and Yumei Yen, to whom this thesis is dedicated, for instilling in me their belief in the value of education.

Table of Contents

CHAPTER 1: INTRODUCTION 1
CHAPTER 2: BACKGROUND 10
2.1. The computational model 10
2.2. Image processing tasks 12
2.2.1. Neighbohood operations 12
2.2.2. Global transforms 15
2.3. Image partitioning 18
CHAPTER 3: A MULTIPLE-PROCESSOR ARCHITECTURE 23
3.1. Introduction 23
3.2. The parallel memory system M 25
3.3. The storage scheme S 25
3.4. The PE's topology T 34
3.5. The interconnection network Θ 37
3.5.1. Data alignment requirements 38
3.5.2. Construction of a theta network 45
3.5.3. Properties of the network 49
3.6. Access and alignment of P_A^I 54
3.7. The machine instructions I 62
3.8. Conclusion 68
CHAPTER 4: ALGORITHMS FOR NEIGHBORHOOD OPERATIONS 69
4.1. Introduction 69
4.2. A parallel algorithm for 3×3 convolution 70
4.3. A parallel algorithm for global convolution 83
4.4. Conclusion 86
CHAPTER 5: ALGORITHMS FOR GLOBAL TRANSFORMS 87
5.1. Introduction 87
5.2. 2-D Fast Fourier Transform 88
5.2.1. A parallel algorithm for 1-D FFT 91
5.2.2. A parallel algorithm for 2-D FFT 99
5.3. Matrix multiplication 100
5.3.1. A sequential matrix multiplication algorithm 101
5.3.2. A parallel matrix multiplication algorithm 102
5.4. Conclusion 106
CHAPTER 6: CONCLUSION 108
References 110

List of Figures

Figure	Page
2.1 A computational model	10
2.2 A system configuration	11
3.1 A multiple-processor configuration	24
3.2 The memory module matrix	27
3.3 An array of PE's for $N = 16$	35
3.4 A simplified architecture of PE	36
3.5 Subimage distribution	39
3.6 Mesh distribution	41
3.7 Row distribution	42
3.8 Column distribution	44
3.9 The theta network for $N = 16$	45
3.10 Eight possible states of a switching element	46
4.1 $PE_{i,j}$ accessing $S_{i,j}^{0,0}$	72
4.2 PE's processing $S_{3,3}$	78
4.3 $PE_{i,j}$ rectifying $S_{i,j}^{3,3}$	80
5.1 Accessing the left two halves of row 0 and row 2	96
5.2 Accessing the right two halves of row 0 and row 2	97

CHAPTER 1

INTRODUCTION

Although the construction of a machine for the understanding of pictures is not yet within reach, the ways in which von Neumann computer architecture falls short of the tasks are already evident [11, 19]. In understanding a simple picture, what the human mind does almost effortlessly takes the fastest existing von Neumann computers many days. Apparently, part of the problem is that the algorithms for understanding general images have not been fully understood. However, part of the problem is the speed restrictions of von Neumann machines. Consider the speed requirement of a sequential computer in keeping up the data flow from standard television image sequences. Suppose that it is required to perform a 3×3 convolution operation on a new image of 512×512 pixels every twenty-fifth of a second. The operation involves, at every pixel, fetching nine pixel values, carrying out eight additions and nine multiplications, and ending with a store operation. On top of that, input and output operations would be also required. Thus, a processing rate of more than 100 million operations per second is needed in this specific problem, even the convolution considered here involves only a small sized convolution kernel. Clearly, a sequential computer is unable to offer processing rates in the range of 100 million to 10 billion operations per second to keep up the data flow cost-effectively. Although a von Neumann machine can be speeded up by further increasing the speeds of circuit components to go some of the way towards enhancing performance, the bulk improvement has to be found elsewhere, especially in pursuing parallelism in the computation. The purely sequential processing nature of the von Neumann computer is one cause of its inefficiency in image processing. The general-purpose

nature of the von Neumann computer is another cause leading to the inefficiency since it implements dissimilar operations with similar computing structure. This is perhaps the von Neumann bottleneck in image processing.

The obvious answer is to abandon von Neumann computer architecture and search for parallel computer architectures to exploit parallelism for typical image processing tasks in the interest of maximizing computational throughput. Parallelism is a loosely defined term which implies that more than one operation of the same sort is performed at each instant, and that the operations are the same kind and proceed in synchronism [48].

The first paper dealing with the problem of matching the computer architecture to the structure of image data was written by Unger [44] in 1958. In his now classical paper, Unger suggested a two-dimensional connected grid of several hundred bit-serial processing elements all under the control of a master processor to solve the problems of pattern recognition [45]. This computing structure is now known as a SIMD (single instruction stream, multiple data stream) machine according to Flynn's classification scheme [17]. In this application, each processing element stores one point of the image. The 512×512 point image would use a quarter million of processing elements. In this case, each step of the convolution operation can be performed locally within a pixel's processing element or through direct communication with the immediate connected neighbors of the processing element. In sequential computers, a substantial amount of time could be spent in address calculation, both in stepping through the pixels as each one is computed and in fetching pixel values from the local neighborhoods. In sharp contrast, this activity is largely unnecessary in SIMD computers [8, 32, 42]. The local neighborhood operation works elegantly on an Unger type computer because the communication structure of the processing elements matches the structure of the problem. Indeed, many of the contemporary SIMD computers

built, under construction, or proposed for image processing reflect precisely the Unger type computer architecture. Well known SIMD machines such as Illiac IV [2], STARAN [4], Cellular Logic Image Processor (CLIP) [10, 11], Distributed Array Processor (DAP) [33], and the Massively Parallel Processor (MPP) with an array of 128×128 processing elements [5] have been actually built, and are in operation today. There are others, e.g., BASE proposed by Reeves [34] and the Connection Machine (CM-1) with 64K processing elements by Hillis [19]. The CM-1 has both 4-connected mesh connections and a powerful interconnection network. As the CM-1 tends to be a general-purpose parallel processor rather than an image processor, its interconnection network is of the packet-switching type [19]. In sharp contrast, all other image processors above use circuit-switching networks [9], where fast and conflict-free routing of image data is the key in achieving image parallelism. As will be seen later, the SIMD architecture proposed in this thesis achieves its flexibility in exploiting image parallelism by making use of a circuit-switching network, which is more powerful than the one used by the CM-1.

Another effective method of coping with some image processing tasks involves the use of Kung's systolic array [21]. Recently, a systematic technique for the optimized mapping of algorithms which can be represented as linear recurrences onto systolic arrays has been proposed [25]. A number of systolic arrays for image processing tasks have been developed [14, 18, 21]. In essence, a systolic array is a tessellated structure of synchronous processors that perform fixed sequences of computations with fixed patterns of communication optimized for a specific application. The optimization trades speed for flexibility. The effective use of systolic arrays in constructing a versatile parallel image processor in an integrated way remains to be further demonstrated. Other classes of architectures, which have been studied and sometimes suggested for image processing

in the literature, include pipelined machines [26], Multi-SIMD machines [40], reconfigurable processor arrays, and MIMD (multiple instruction stream, multiple data stream) machines [12, 27]. They do not appear to be particularly cost-effective approaches, however, for well-structured problems with regular patterns of control where image processing at the pixel level is the case in point. By contrast, the SIMD machine can devote most of its hardware to operations on the image data, and share most of its control hardware among all the processing elements [42, 48].

It is possible today to build a SIMD computer with hundreds of thousands of simple processing elements which has raw computing power that is many orders of magnitude greater than the fastest existing conventional computers [5, 15, 19, 28, 36]. The problem lies in how to couple the raw computing power with the image processing tasks of interest. In an attempt to classify image processing tasks with a basic general operation, Danielsson and Levialdi [9] have considered the neighborhood operation as typical. It has been demonstrated that the regular structures of image data and neighborhood operations can be mapped most naturally onto SIMD architectures, where the identical transformation to be carried out on every pixel can be done for as many pixels as there are processing elements. In principle, if the number of processing elements is the same as the number of pixels, the throughput rate is independent of the size of the image [37]. This performance is, however, rarely achieved in practice, due to the practical constrain that the number of processing elements cannot be arbitrarily large in any physical machine. A fundamental problem in the design of SIMD processor arrays for image processing, which naturally arises from the limitations on the number of processing elements, is concerned with how to partition the image into smaller parts that can be processed by the processor array on the one processor per pixel basis with a high degree of parallelism [9, 13].

From the standpoint of functionality, a parallel image processor is essentially a hardware implementation of the partition strategy chosen for the image processing tasks of interest. In the heart is the hardware that is able to move these parts between the processor array and the place where they are stored. There are two commonly used partition strategies. The first one involves partition of the image into subimages and uses the processor array to work on one subimage at a time in such a way that neighboring pixels of the subimage reside in neighboring processing elements [9]. Clearly, the parallelism is achieved by allowing all processing elements to work on a subimage simultaneously, although each of the subimages is processed in sequence. This is the partition strategy adopted by both CLIP IV and MPP. The advantage of such a strategy follows directly from the Unger computer architecture in algorithm-architecture matching for 3×3 neighborhood operations. It is obvious that each nonedge processing element that is not located at the edge of the processor array can access its immediate neighbors directly to obtain information necessary for the transformation on its responsible pixel simultaneously. However, one difficult problem to overcome here is to handle the pixels on the boundaries between subimages. The required pixel values have to be transferred with additional and selective access cycles, thereby increasing the overhead and losing parallelism. Moreover, access to a larger neighborhood requires shifting of data among the processing elements, which is not only a nontrivial problem but also slows down the processing speed considerably for sparsely filled neighborhood kernels with many 0's [9].

The second strategy overcomes these two problems, to a great extent, by distributing the processing elements evenly over the entire image in such a way that each subimage is assigned to a separate processing element and neighboring subimages are assigned to neighboring processing elements. With this strategy, while each processing element working on its subimage sequentially all subim-

ages are processed by the processor array in parallel. It was first suggested for the DAP machine using its so-called row and column highways to convert the image data into this format. This scheme is now adopted in the design of the image processor LIPP [13]. The second method solves the edge problem far more smoothly, and makes it possible to access larger neighborhoods with ease and efficiency. However, the cost is that it cannot take the advantage of using the communication topology of the processing elements to perform 3×3 neighborhood operations. Yet, operations localized on a small number of subimages can not be performed as efficiently.

From the algorithm-architecture matching point of view, the first strategy offers an excellent computing structure to exploit parallelism in executing local neighborhood operations. In contrast, the second strategy provides a pertinent computing structure to exploit parallelism in performing larger neighborhood operations. Each of the resulting image processors is, thereby, able to couple the computational power with the image processing tasks of interest.

Although neighborhood operations are considered typical, global transforms such as two-dimensional Fast Fourier Transform (2-D FFT) deserves an important place in image processing as a fast computation technique [38]. Computationally, global transforms involve global communication of the unstructured data such as 2-D FFT and matrix multiplication where global communication is the key property of the algorithms [1]. Unfortunately, none of the two partition strategies mentioned above could offer algorithm-architecture matching for operations in this category. Parallel implementation of the FFT algorithm on SIMD machines has been studied by a number of researchers [4, 20, 30, 41], among others. In particular, the STARAN associative processor [4] can support the FFT algorithm at the maximum efficiency. In STARAN, the image is partitioned into rows and columns, and a 1-D set of processing elements are used to work on one

row or one column at a time in parallel using its Flip network to achieve the required global data communication. However, the weakness of STARAN in image processing applications lies in that it does not provide a suitable computing structure for neighborhood operations, especially for larger neighborhood operations. It is widely believed that future parallel computers for image processing at the pixel level should possess more general and hence more flexible architecture on which a variety of image processing algorithms can be efficiently implemented [9, 22, 35, 37, 47].

Consequently, this research is aimed at the development of a flexible parallel computer architecture for image processing at the pixel level. The approach taken in achieving such a goal is primarily motivated by the following observations. First, the efficiency of a parallel algorithm, in general and for image processing in particular, depends on the particular parallel architecture chosen for its execution. Second, for each image processing algorithm, extreme parallelism can be exploited in the computation so long as the structure of the algorithm matches the structure of the computer on which it is implemented. Third, the partition strategy embodied in an image processor plays an important role in achieving algorithm-architecture matching for the intented class of image processing algorithms. Finally, an image processor that implements more useful partition strategies should be more general and hence more flexible as long as the partition strategies can be used to couple the computational power of the computer with the application of interest. Accordingly, the parallel architecture possesses the following feature. A set of partition strategies is embodied in the architecture so that algorithm-architecture matching can be dynamically achieved according to the image processing algorithm to be executed. The two partition strategies discussed previously are two members of this set. The parallel architecture is flexible enough in performing with a high degree of parallelism

not only neighborhood operations, but also operations in the category of global transforms, and hence the generality.

The thesis begins with a discussion of a two-level computational model for image processing and computational structures of some low-level image processing tasks, in Chapter 2. Definitions and terminology are introduced in preparation for the development and explanation of the parallel computer architecture.

A multiple-processor architecture of the SIMD type consisting of N processing elements and N memory modules is introduced in Chapter 3. The proposed architecture implements a set of N partition strategies for the purpose of algorithm-architecture matching. As a result, dissimilar image processing operations can be implemented with dissimilar constructs on the architecture as opposed to von Neumann computers which implement dissimilar operations with similar construct. Such an architectural feature is achieved by making use of a novel storage scheme in conjunction with a fast multistage interconnection network. Functionally, the storage scheme defines the set of partition strategies, and the interconnection network moves partitioned parts between processing elements and memory modules. Fundamental properties of the computer architecture are given, and some of the implementation considerations are provided.

Based on the architecture proposed in Chapter 3, Chapter 4 discusses the problems of efficient implementation of neighborhood operations. Choosing pertinent partition strategies which in turn determine algorithm-architecture matching, the architecture can execute both local and global neighborhood operations with a high degree of parallelism. The use of the proposed architecture for pyramid processing [7, 43] is also discussed.

Chapter 5 deals with the problem of efficient implementation of global transforms on the proposed architecture. In particular, two parallel algorithms are presented for the 2-D FFT and matrix multiplication, respectively. Again,

the key to exploit parallelism is the use of appropriate partition strategies.

The last chapter of this thesis is a brief conclusion and a discussion of possible directions for future research.

CHAPTER 2

BACKGROUND

This chapter establishes the fundamentals of a parallel computer architecture for image processing which will be used in subsequent chapters: the computational model adopted, the characteristics of three classes of typical image processing tasks, and image partitioning strategies which lead to parallelism in the computation of image processing tasks.

2.1. The computational model

The computational model adopted in this thesis for image processing consists of two levels as shown in Fig. 2.1. The model is widely used in the literature.

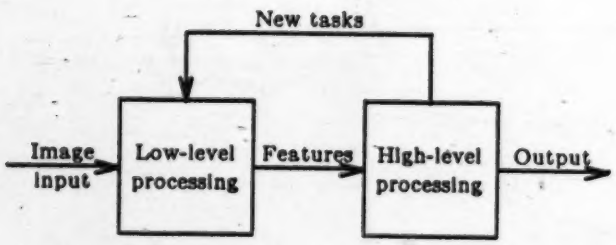

Fig. 2.1. A computational model.

The low-level component performs low-level image processing tasks which could be called filtering [38] in the most general sense. In addition, it also performs functions like image input/output, delivering feature data to the high-level component. The feature data could be, for example, a histogram of an image.

The high-level component evaluates feature data, resulting in a final output or a new low-level image processing task fed back to the low-level component. For example, it processes the histogram to determine a threshold [38].

This model assumes that the high-level is in control of the low-level. Low-level image processing tasks are usually computationally intensive. In contrast, high-level involves a lot of decision making but little computation.

Corresponding closely to this computational model, the computer system assumed in this thesis consists of two components: a parallel computer and a sequential computer, as depicted in Fig. 2.2.

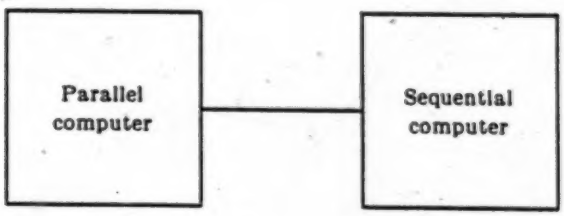

Fig. 2.2. A system configuration.

The parallel computer is a multiple-processor to be introduced in Chapter 3, and performs the bulk of the computation, i.e., the low-level image processing tasks. The sequential computer is a general-purpose host computer, and plays the role of high-level processing. It is responsible for executing tasks of the high-level type as well as feeding input images to the parallel computer. This configuration is flexible. It allows the host to do some low-level image processing tasks that are not suited to the parallel computer.

The architecture of the parallel computer is the main subject of this study, and will be fully developed in subsequent chapters. The host computer will not

be considered any further. The low-level image processing tasks will be simply called image processing tasks hereafter.

2.2. Image processing tasks

The input data to the low-level component are always in the form of two-dimensional matrices called images, as opposed the data input to the high-level component of the computational model which has no unified structure. The quantity of data to be processed by the low-level component is large, which is a ready source of parallelism. In addition, image processing tasks, to a large extent, lend themselves to parallelism. Three classes of computationally intensive image processing tasks will be considered. They are local neighborhood operations, global neighborhood operations, and global transforms, respectively. The multiple-processor must be effective for performing at least image processing tasks within these three classes.

2.2.1. Neighbohood operations

Neighborhood operations are widely used in image processing and pattern recognition for smoothing, deblurring, edge detection, texture analysis, etc. [38], and are, therefore, considered as one of the most important classes of operations [9]. Throughout this thesis, let matrix $\mathbf{P}$ represent an $N \times N$ array of picture elements, or pixels:

$$\mathbf{P} = \begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,N-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,N-1} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ p_{N-1,0} & p_{N-1,1} & \cdots & p_{N-1,N-1} \end{pmatrix},$$

where each element $p_{i,j}$ for $i,j \in \{0,1, \ldots, N-1\}$ is an integer that represents the intensity, or gray-level of a portion of the image.

As a result of applying a neighborhood operation to **P**, a new image **G** is computed

$$\mathbf{G} = \begin{pmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,N-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,N-1} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ g_{N-1,0} & g_{N-1,1} & \cdots & g_{N-1,N-1} \end{pmatrix}.$$

The neighborhood operation computes each $g_{i,j}$ as a function of a set of pixels of **P** in a neighborhood relating to pixel $p_{i,j}$. In other words, each new pixel will have a gray-level which is calculated from pixels in a window either surrouding or near to the corresponding pixel in the original image. The size and shape of the window may be varied, and it is not implied that all functions will make use of every pixel in the window.

There are several ways in which neighborhood operations can be classified. One possibility would be to draw up a functional classification in terms of the new image **G** obtained by applying a neighborhood operation to **P**. It is, how-

ever, more instructive to classify neighborhood operations with respect to window dimensions, which determine the pixels included in the computation. Such a classification is especially informative from the computer architecture point of view. This is because it suggests a catalogue of useful inter-processor communication structures that an image processing computer may want to have in order to match the communication structures of the neighborhood operations to fully exploit the inherent parallelism.

The most widely used classification scheme is one in which neighborhood operations are classified into two categories: local neighborhood operations and global neighborhood operations [11]. In the category of local neighborhood operations, a 3×3 window is used by each operation, while in the category of global neighborhood operations, a window of size larger than 3×3 is employed. In other words, a local neighborhood operation is one in which $g_{i,j}$, the output value, is a function of $p_{i,j}$ and directly adjacent pixels of $p_{i,j}$ in $\mathbf{P}$.

For example, the convolution of the image $\mathbf{P}$ and an $M \times M$ array $\mathbf{W}$, which is called the convolution kernel, given by

$$g_{i,j} = \frac{1}{M^2} \sum_{h=0}^{M-1} \sum_{v=0}^{M-1} w_{h,v} \times p_{i-h,j-v},$$

where $i,j \in \{0,1, ..., N-1\}$, is a neighborhood operation. The values of $p_{h,v}$ are assumed to be zero when the indices are not in $\{0,1, ..., N-1\}$. With this condition, the convolution defined is called a linear convolution. Convolution operations thus fall into two different categories of neighborhood operations according to the size of the convolution kernel.

Clearly, with global neighborhood operations in which each output depends on any or a large portion of the input, there is communication among distant parts of the image. The computation of the convolution requires $O(N^2 \times M^2)$ multi-

plications and additions. As M becomes large, convolution can be very costly.

2.2.2. Global transforms

Global transforms are operations in which each output depends on any portion of the input image. Examples of global transforms are 2-D Fourier transform, Hadmard transforms, and matrix manipulations such as matrix multiplication [38]. Two operations in this class will be considered in this thesis. They are 2-D FFT and matrix multiplication. The Fourier transform arises in many fields of science and engineering, and a fast method for computing the Fourier transform has long been of great interest [1]. One of the major applications of the 2-D FFT is in computing the convolution of two images **P** and **W**, by making use of the Convolution Theorem [38]. In this application, it is computationally efficient to first apply 2-D FFT to image **P** and image **W**, then compute a new image by multiplying pairs of the corresponding elements in these two transformed images, and finally to apply the inverse 2-D FFT to the new image to get the desired convolution. When M becomes as large as N, the direct convolution method requires $O(N^4)$ arithmetic operations. However, the use of the 2-D FFT will reduce the cost to $O(N^2 \log N)$.

The 2-D Discrete Fourier Transform (DFT) of image **P** is defined as follows:

$$g_{k_1,k_2} = \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} p_{n_1,n_2} W^{k_1 n_1} W^{k_2 n_2},$$

where W is a primitive Nth root of unity.

It can be rewritten as:

$$g_{k_1,k_2} = \sum_{n_1=0}^{N-1} \left(\sum_{n_2=0}^{N-1} p_{n_1,n_2} W^{k_2 n_2} \right) W^{k_1 n_1},$$

which indicates the 2-D DFT of image **P** can be computed by decomposing it

into row and column 1-D DFT's, i.e., computing the 1-D DFT of each row of P and storing the result in an intermediate array, then computing the 1-D DFT of each column of the intermediate array.

The problem of computing an N point 1-D DFT is as follows. Given $x(0)$, $x(1)$, ..., $x(N-1)$, compute $X(0)$, $X(1)$, ..., $X(N-1)$ defined by

$$X(i) = \sum_{k=0}^{N-1} x(k) W^{ik} \tag{1}$$

The well known 1-D FFT algorithm solves an N point DFT problem in $O(N \log N)$ operations, while the straightforward method requires $O(N^2)$ operations. The 1-D FFT method achieves its computational efficiency through a "divide and conquer" strategy [1]. If the DFT length is, for example, a power of 2, i.e., $N = 2^n$, the DFT can be expressed as a combination of the half-length DFT's, each of which can be expressed in turn as a combination of two quarter-length DFT's, and so on. The decimation-in-time version of the 1-D FFT algorithm is derived most directly by expressing the indices i and k in the binary representation [30, 41].

Substituting the identities

$$i = i_0 i_1 \cdots i_{n-1}$$

and

$$k = k_0 k_1 \cdots k_{n-1}$$

into (1) yields

$$X(i_0 \cdots i_{n-1}) = \sum_{k_{n-1}} \sum_{k_{n-2}} \cdots \sum_{k_0} x(k_0 \cdots k_{n-1}) W^{\left(\sum_{m=0}^{n-1} i_m 2^{n-m-1}\right)\left(\sum_{m=0}^{n-1} k_m 2^{n-m-1}\right)} \tag{2}$$

Define n vectors T_s, $1 \le s \le n$, as follows:

$$T_1(i_{n-1} k_1 \cdots k_{n-1}) = \sum_{k_0} x(k_0 \cdots k_{n-1}) W^{i_{n-1} k_0 2^{n-1}},$$

and, for $2 \leq s \leq n$,

$$T_s(i_{n-1} \cdots i_{n-s} k_s \cdots k_{n-1})$$
$$= \sum_{k_{s-1}} T_{s-1}(i_{n-1} \cdots i_{n-s+1} k_{s-1} \cdots k_{n-1}) W^{(\sum_{m=n-s}^{n-1} i_m 2^{n-m-1}) k_{s-1} 2^{n-s}} \tag{3}$$

Using the relation

$$W^{(\sum_{m=0}^{n-1} i_m 2^{n-m-1}) k_{s-1} 2^{n-s}} = W^{(\sum_{m=n-s}^{n-1} i_m 2^{n-m-1}) k_{s-1} 2^{n-s}}$$

in the summation of (3), it is easy to show

$$T_n(i_{n-1} \cdots i_0)$$
$$= \sum_{k_{n-1}} \sum_{k_{n-2}} \cdots \sum_{k_0} x(k_0 k_1 \cdots k_{n-1}) W^{(\sum_{m=0}^{n-1} i_m 2^{n-m-1})(\sum_{m=0}^{n-1} k_m 2^{n-m-1})}$$

$$= X(i_0 \cdots i_{n-1}).$$

To obtain the value of $X(i)$ when given vector T_n, it is necessary to reverse the binary digits in the binary representation of i to get a new index $\bar{i}$. Then, $X(i) = T_n(\bar{i})$.

In the above formulation, each element of T_s is the weighted sum of two elements of T_{s-1}. To be more precise, $T_s(i_0 \cdots i_{n-1})$ is the weighted sum of $T_{s-1}(i_0 \cdots i_{s-2} 0 i_s \cdots i_{n-1})$ and $T_{s-1}(i_0 \cdots i_{s-2} 1 i_s \cdots i_{n-1})$.

Now, consider the two elements of T_s whose indices differ only in the $(s-1)$th bit position,

$$T_s(i_{n-1} \cdots i_{n-s+1} 0 k_s \cdots k_{n-1})$$
$$= T_{s-1}(i_{n-1} \cdots i_{n-s+1} 0 k_s \cdots k_{n-1})$$
$$+ T_{s-1}(i_{n-1} \cdots i_{n-s+1} 1 k_s \cdots k_{n-1}) W^{(\sum_{m=n-s+1}^{n-1} i_m 2^{n-m-1}) 2^{n-s}} \tag{4}$$

and

$$T_s(i_{n-1} \cdots i_{n-s+1} 1 k_s \cdots k_{n-1})$$
$$= T_{s-1}(i_{n-1} \cdots i_{n-s+1} 0 k_s \cdots k_{n-1})$$

$$+ T_{s-1}(i_{n-1} \cdots i_{n-s+1} 1 k_s \cdots k_{n-1}) W^{\left(\sum_{m=n-s+1}^{n-1} i_m 2^{n-m-1} + 2^{s-1}\right) 2^{n-s}}$$

$$= T_{s-1}(i_{n-1} \cdots i_{n-s+1} 0 k_s \cdots k_{n-1})$$

$$- T_{s-1}(i_{n-1} \cdots i_{n-s+1} 1 k_s \cdots k_{n-1}) W^{\left(\sum_{m=n-s+1}^{n-1} i_m 2^{n-m-1}\right) 2^{n-s}} \qquad (5)$$

The computation defined in (4) and (5) is called the butterfly operation that computes the two elements of T_s from the two elements of T_{s-1} with one multiplication and two additions.

The butterfly operation is the key communication structure of the FFT algorithm. As can be seen, it is entirely different from that of neighborhood operations.

2.3. Image partitioning

As images can range from the 480×640 size of standard television up to a 15,000×15,000 array of typical weather satellites, it is unlikely that an image processor will ever have enough processing elements to address the largest practical image. In other words, image partitioning is inevitable. In image partitioning, the image is decomposed into smaller parts, and the image processor processes one part at a time, and generates the output corresponding to that part. The outputs are then combined. The overhead in the computation time for image partitioning comes from three souces: first, the cost of moving the parts between the image processor and the place in which they are stored; second, the cost of keeping any additional information to allow outputs to be recombined later; and third, the cost of recombining the outputs. A good image partitioning strategy should minimize the three costs simultaneously for image processing tasks of interest. The image partitioning strategy is considered in this thesis as an important concept in image processor design in that an image processor is

essentially a hardware implementation of the partitioning strategy(s) chosen for the intended image processing tasks.

It is convenient sometimes, particularly in discussing image partitioning strategies, to consider the image $\mathbf{P}$ as a set of pixels. For this reason, the symbol $\mathbf{P}$ will also be used to denote the set of all picture elements of the image, i.e., $\mathbf{P} = \{p_{i,j} \mid 0 \leq i,j \leq N-1\}$. To discuss image partitioning more precisely, the following teminology will be used in this thesis.

Definition: Define the union of N sets as follows:

$$\bigcup_{k=0}^{N-1} \Omega_k = \Omega_0 \cup \Omega_1 \cup \cdots \cup \Omega_{N-1}$$

$$= \{a \mid a \in \Omega_k \text{ for some } 0 \leq k \leq N-1\}.$$

Definition: Any finite number of sets $\Omega_0, \Omega_1, \ldots, \Omega_{N-1}$ are said mutually disjoint if $\Omega_i \cap \Omega_j = \varnothing$ for all $i \neq j$.

Definition: A collection $\Pi = \{\Omega_0, \Omega_1, \ldots, \Omega_{N-1}\}$ of nonempty subsets $\Omega_k \subset \mathbf{P}$ is called a partition of $\mathbf{P}$ if

1) $\bigcup_{k=0}^{N-1} \Omega_k = \mathbf{P}$,

2) $\Omega_i \cap \Omega_j = \varnothing$ for all $i \neq j$.

Definition: A partition of $\mathbf{P}$ into N equal sized parts (subsets) is called an N-partition of image $\mathbf{P}$.

The importance of an N-partition of image $\mathbf{P}$ becomes clear in the context of using N processing elements to perform image processing tasks on $\mathbf{P}$. It is apparent that any part of an N-partition can be processed by the processing elements on the basis of one processing element per pixel. Obviously, an N-partition of images does not necessarily provide a good image partitioning strategy in

terms of minimizing the overhead discussed earlier. An N-partition of $\mathbf{P}$ does not convey any information about how the image is decomposed unless each part is defined. A set of N-partitions of image $\mathbf{P}$ will be described in Chapter 3, where each part of an N-partition is defined.

The remainder of this chapter will illustrate four typical N-partitions of image $\mathbf{P}$, and introduce some notation to be used in subsequent chapters.

Let $N = 2^n$, and denote by $\mathbf{I}_N$ the set $\{0,1, ..., N-1\}$. The elements of $\mathbf{I}_N$ will be considered as integers, residue classes modulo N, or n-dimensional vectors whose components are either 0 or 1. In case of the vector $x_0 \cdots x_{n-1}$, it is identified with the integer $x = \sum_{i=0}^{n-1} x_i 2^{n-i-1}$, with x_0 being the most significant bit. The vector $x_0 \cdots x_{n-1}$ is also called the binary representation of the integer x. Futhermore, let the integer n be an even number. This implies the integer N is a power of four.

Let r be an integer and $r = \sqrt{N}$. Denote by $\mathbf{I}_r$ the set $\{0,1, ..., r-1\}$. Next, define $\Gamma(i)$ and $\Omega(i)$ to be two integer functions $\Gamma(i) = \lfloor \frac{i}{r} \rfloor$, and $\Omega(i) = i \bmod r$, respectively, where $i \in \mathbf{I}_N$ and $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

Definition: For $i \in \mathbf{I}_N$, define Ω_i to be a set of pixels of image $\mathbf{P}$, $\Omega_i = \{p_{a+x,b+y} \mid x,y \in \mathbf{I}_r,\ a = r \times \Gamma(i),\ b = r \times \Omega(i)\}$. Ω_i will be called the ith subimage of image $\mathbf{P}$.

In fact, the collection of Ω_i is an N-partition of image $\mathbf{P}$. These N subimages have a spatial relationship among them in the following sense. Decomposing image $\mathbf{P}$ into an $r \times r$ array of subimages, the ith subimage is found at row $\Gamma(i)$ and column $\Omega(i)$ of the array. Each subimage has eight immediate neighboring subimages except those located at the edges of the array. For description convenience, subimage i and subimage $(\Gamma(i),\Omega(i))$ will be used to refer to the same

subimage, interchangeably.

Similarly, pixels of the ith subimage (Ω_i) of image **P** have a spatial ordering. Organizing the pixels of Ω_i as an $r \times r$ array, pixel $p_{a+x,b+y}$ is found at row x and column y of the array. The pixel located at row i and column j of a subimage will be called pixel (i,j) of that subimage, where $i,j \in I_r$. Each pixel of a subimage has eight immediate neighbors in the subimage except those located at the borders.

Definition: For $i \in I_N$, define Ω_i to be a set of pixels of image **P**, $\Omega_i = \{p_{a+x \times r, b+y \times r} \mid x,y \in I_r,\ a = \Gamma(i),\ b = \Omega(i)\}$. Ω_i will be called the ith mesh of image image **P**.

The collection of Ω_i is also an N-partition of image **P**. The pixels of a mesh have spatial ordering. Arranging them as an $r \times r$ array, pixel $p_{a+x \times r, b+y \times r}$ of Ω_i is found at row x and column y of the array.

The pixel located at row i and column j of a mesh will be referred to as pixel (i,j) of that mesh, where $i,j \in I_r$. Each pixel of a mesh has eight neighbors in that mesh except those located at the edges of the mesh. The neighbors have, however, a larger distance from each other than the immediate neighbors, i.e., r versus 1. The following relationship will be frequently used in later discussions: pixel (i,j) of mesh k is pixel $(\Gamma(k),\Omega(k))$ of subimage (i,j).

Definition: For $i \in I_N$, define Ω_i to be a set of pixels of image **P**, $\Omega_i = \{p_{i,x} \mid x \in I_N\}$. Ω_i will be called the ith row of the image.

Clearly, the collection of Ω_i is an N partition of the image. Pixel $p_{i,x}$ will be referred to as the xth pixel of the row.

Definition: For $i \in I_N$, define Ω_i to be a set of pixels of image **P**, $\Omega_i = \{p_{x,i} \mid x \in I_N\}$. Ω_i will be called the ith column of the image.

The collection of Ω_i is also an N-partition of the image. Pixel $p_{x,i}$ will be refered to as the xth pixel of the column.

CHAPTER 3

A MULTIPLE-PROCESSOR ARCHITECTURE

This chapter will present a multiple-processor architecture of the SIMD type for image processing at the pixel level. It has been recognized that the efficiency of a parallel algorithm depends on the particular parallel architecture chosen for its execution. Ideally, the computational structure of the algorithm should match that of the machine on which it is implemented. The proposed computer architecture is able to support the structural matching dynamically, and is thereby capable of exploiting a high degree of parallelism and efficiency for a variety of image processing and pattern recognition algorithms.

Fundamental properties of the architecture are described, and some implementation considerations also provided. Theorems insuring the validity of the architecture are proven.

3.1. Introduction

The proposed multiple-processor architecture is of SIMD type, and consists of a set of N PE's (processing elements), a set of N memory modules, an interconnection network, and a CUN (control unit), as illustrated in Fig. 3.1, for processing an image of dimensions $N \times N$.

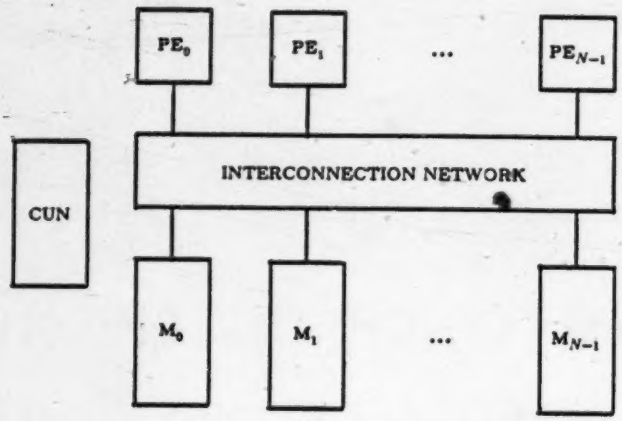

Fig. 3.1. A multiple-processor configuration.

Being of the SIMD type, the single instruction stream is fetched and decoded by the control unit. Control unit instructions are also executed there. Processing element instructions are broadcast to the PE's, and each PE simultaneously executes the same instruction on its own appropriate data. The interconnection network allows the transfer of information among PE's, or the alignment of data between PE's and memory modules in a deterministic and short amount of time.

The multiple-processor architecture for processing image **P** is now formally defined as the 6-tuple **(N, M, S, T, Θ, I)** where:

1) **N** is an integer representing the number of PE's,

2) **M** is a parallel memory system,

3) **S** is a storage scheme,

4) **T** is a topological configuration of the PE's,

5) **Θ** is an interconnection network,

6) $\mathbf{I}$ is a set of machine instructions.

The motivation for formalizing the proposed computer architecture lies in the need of a formal model for representing the architecture and analyzing its behavior in a precise way. The integer $\mathbf{N}$ is the same N defined previously in Chapter 2, and needs no further comment. The remaining five components of the architecture will be presented in subsequent sections along with their fundamental properties and some implementation details. Definitions and terminology introduced in Chapter 2 are used freely. Additional definitions will be introduced as needed.

3.2. The parallel memory system M

The parallel memory system $\mathbf{M}$ consists of N memory modules denoted

$$\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_{N-1}$$

where $\mathbf{M}_i$ for $i \in \mathbf{I}_N$ denotes the ith memory module, and has a memory capacity of N, which represents the total number of picture elements of $\mathbf{P}$ the module is able to store. Collectively, $\mathbf{P}$ can be stored in $\mathbf{M}$. With this memory system it is possible to access N picture elements simultaneously, one in each memory module. Hereafter, $\mathbf{M}_{i,j}$ is used to denote the jth location of $\mathbf{M}_i$ for $j \in \mathbf{I}_N$.

3.3. The storage scheme S

Thinking of $\mathbf{M}$ as a set of memory locations, one can view a storage scheme as an one-to-one and onto mapping of the set $\mathbf{P}$ into the set $\mathbf{M}$. In other words, for any given element of $\mathbf{P}$, a storage scheme determines the memory module and the location of that module where the element will be stored such that each element of $\mathbf{P}$ will reside in exact one memory location of $\mathbf{M}$. In essense, a storage scheme is an arbitrary one-to-one and onto mapping of $\mathbf{P}$ into $\mathbf{M}$; however, the

interesting storage schemes are those that provide useful results.

Define $\xi(i)$ to be an integer function $\xi(i) = r \times i - (N-1)\lfloor \frac{i}{r} \rfloor$, where $i \in \mathbf{I}_N$. More explicitly, the function $\xi(i)$ can be written as follows when the integer i is in binary representation:

$$\xi(i_0 \cdots i_{\frac{n}{2}-1} i_{\frac{n}{2}} \cdots i_{n-1}) = i_{\frac{n}{2}} \cdots i_{n-1} i_0 \cdots i_{\frac{n}{2}-1}.$$

It simply swapps the contents of the higher order $\frac{n}{2}$ bits with that of the lower order $\frac{n}{2}$ bits of its integer argument i. Let $\bar{A}$, AB, and $A \oplus B$ denote the bitwise COMPLEMENT of A, the bitwise AND of A and B, and the bitwise EXLUSIVE-OR of A and B, respectively, where A and B are integers in $\mathbf{I}_N$. The operators $\oplus$, AND, and COMPLEMENT are assumed to have increasing operator precedence. The storage scheme **S** is now presented, and its virtue will be elaborated.

Storage scheme **S**: Store picture element $p_{i,j}$ for $i,j \in \mathbf{I}_N$ in location j of the memory module $\xi(i) \oplus j$.

It can be shown that the storage scheme **S** is an one-to-one and onto mapping of **P** into **M**. As an example, Fig. 3.2 illustrates the storage scheme **S** for the case of $N = 16$, where the entry at the ith row and the jth column of the matrix referred to as the memory module matrix has the value of $\xi(i) \oplus j$ for $i,j \in \mathbf{I}_N$, thereby representing the memory module within which picture element $p_{i,j}$ is stored. Although the memory module matrix does not show explicitly the location within the memory module in which $p_{i,j}$ is stored, it should be clearly understood from the storage scheme **S** that the jth location of the memory module is automatically implied. For instance, Fig. 3.2 indicates that picture element $p_{2,1}$ is stored in location 1 of memory module 9, since the entry at row 2 and column 1 of the memory module matrix has the value of 9.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3
1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10
9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6
13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2
2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13
6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9
10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5
14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1
3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12
7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fig. 3.2. The memory module matrix.

A careful inspection of the storage scheme **S**, in general and the example of Fig. 3.2 in particular, will reveal that the logic order of picture elements of **P** is no longer preserved when they are stored in **M** in the sense that adjacent pixels are not necessarily in adjacent memory modules and adjacent memory locations.

One of the most important factors in determining the performance of a multiple-processor is the system's capability for providing data at the rate matched to the processor rate. It is not sufficient to provide a memory system whose raw bandwith is matched to the bandwith of the N processing elements. Therefore, it is necessary to insure that data access normally occurs without memory conflicts, and each of the N data elements can be brought into the correct processing element for processing. The latter condition implies that means must be provided to restore or change the ordering of the N picture ele-

ments before they can be processed by the N PE's.

Definition: Let Δ be a set of N picture elements of $\mathbf{P}$. If the elements of Δ lie in distinct memory modules when stored in $\mathbf{M}$ under the storage scheme $\mathbf{S}$ then Δ is said to be parallel accessible.

It is clear that Δ can be accessed without memory conflicts if and only if Δ is parallel accessible. Two questions arise at this point. First, does the storage scheme $\mathbf{S}$ provide such parallel accessible sets that suite parallel algorithms for image processing? Second, how can each picture element of a parallel accessible set be brought into the right PE for subsequent processing? This later question will be discussed shortly. The first question concerns the existance of meaningful parallel accessible sets with respect to image processing. The following definition is needed.

Definition: $\mathbf{P}_A^l$, for $l,A \in \mathbf{I}_N$, is defined to be a set of picture elements of image $\mathbf{P}$, $\mathbf{P}_A^l = \{p_{(l\oplus Ak),l\oplus Ak} \mid k \in \mathbf{I}_N\}$.

The $\mathbf{P}_A^l$ defined possesses a number of interesting properties which will be discussed in this section. The following theorem shows that various N-partitions of image $\mathbf{P}$ can be specified in terms of $\mathbf{P}_A^l$.

Theorem 3.1: For any integer $A \in \mathbf{I}_N$, there exists a collection $\Pi = \{\mathbf{P}_A^l \mid l \in \mathbf{I}_N\}$, and Π is an N-partition of image $\mathbf{P}$.

Proof: The proof proceeds by showing the following three statements are true: 1) $|\mathbf{P}_A^l| = N$; 2) $\mathbf{P}_A^i \cap \mathbf{P}_A^j = \varnothing$ for all $i \neq j$; and 3) $\bigcup_{l=0}^{N-1} \mathbf{P}_A^l = \mathbf{P}$.

To prove 1) needs to show that no two elements in $\mathbf{P}_A^l$ correspond to the same picture element in $\mathbf{P}$. Let i and j be any two integers such that $i,j \in \mathbf{I}_N$ and $i \neq j$, then $p_{(l\oplus Ai),l\oplus Ai}$ and $p_{(l\oplus Aj),l\oplus Aj}$ are both in $\mathbf{P}_A^l$. Suppose they

correspond to the same picture element of $\mathbf{P}$, then the following condition holds:

1. $\xi(I \oplus \bar{A}i) \oplus \xi(I \oplus \bar{A}j) = 0$, and

2. $I \oplus Ai \oplus I \oplus Aj = 0$.

Thus, both $\bar{A}(i \oplus j)$ and $A(i \oplus j)$ must be equal to 0. Therefore, i must be equal to j, which is a contradiction.

The proof of 2) is as follows. Let i and j be any two integers such that $i,j \in \mathbf{I}_N$ and $i \neq j$, and let k_1 and k_2 be any two integers satisfying $k_1, k_2 \in \mathbf{I}_N$. Thus, $p_{\xi(i \oplus \bar{A}k_1), i \oplus Ak_1}$ and $p_{\xi(j \oplus \bar{A}k_2), j \oplus Ak_2}$ are in $\mathbf{P}_A^i$ and $\mathbf{P}_A^j$, respectively. Suppose they correspond to the same picture element in $\mathbf{P}$, then the following condition must hold:

1. $\xi(i \oplus \bar{A}k_1) \oplus \xi(j \oplus \bar{A}k_2) = 0$, and

2. $i \oplus Ak_1 \oplus j \oplus Ak_2 = 0$.

This implies that i must be equal to j, which is a contradiction.

The correctness of the third statement is an immediate consequence of the first two. Since each $\mathbf{P}_A^I$ corresponds exactly to N elements of $\mathbf{P}$, and all $\mathbf{P}_A^I$ are mutually disjoint, hence, the union is $\mathbf{P}$.

Q.E.D.

The next theorem shows that if image $\mathbf{P}$ is stored in $\mathbf{M}$ according to the storage scheme $\mathbf{S}$, then the N elements of $\mathbf{P}_A^I$ are uniformly distributed among the N memory modules.

Theorem 3.2: Set $\mathbf{P}_A^I$ is parallel accessible.

Proof: Let pixels $p_{\xi(I \oplus \bar{A}i), I \oplus Ai}$ and $p_{\xi(I \oplus \bar{A}j), I \oplus Aj}$ be any two elements of $\mathbf{P}_A^I$, where $i,j \in \mathbf{I}_N$. According to the storage scheme $\mathbf{S}$, $p_{\xi(I \oplus \bar{A}i), I \oplus Ai}$ is stored in memory module $\xi(\xi(I \oplus \bar{A}i)) \oplus (I \oplus Ai)$, which equals to i. Likewise, $p_{\xi(I \oplus \bar{A}j), I \oplus Aj}$ is stored in memory module $\xi(\xi(I \oplus \bar{A}j)) \oplus (I \oplus Aj)$, which equals to j. That is, $p_{\xi(I \oplus \bar{A}i), I \oplus Ai}$ and

$P_{(i\oplus j),i\oplus A_j}$ are stored in M_i and M_j, respectively.

Since they are two different elements of P_A^l, the condition $i \neq j$ holds due to Theorem 3.1. This implies that all elements of P_A^l are in distinct memory modules.

Q.E.D.

The two theorems given above have several important consequences. Firstly, Theorem 3.1 indicates that for any fixed integer $A \in I_N$, there exists an N-partition of $\mathbf{P}$ of which each part is defined precisely by a P_A^l for some $l \in I_N$. Therefore, processing these parts one at a time will ultimately end up with covering the whole image $\mathbf{P}$. Secondly, Theorem 3.2, on the other hand, guarantees that any of these parts can be accessed from the parallel memory system $\mathbf{M}$ without memory conflicts. This simply means that when $\mathbf{P}$ is stored in $\mathbf{M}$ under the storage scheme $\mathbf{S}$, the bandwith of the parallel memory system $\mathbf{M}$ can be matched to the bandwith of the N PE's as far as accessing P_A^l goes.

It is important to observe that there exist N possible N-partitions of $\mathbf{P}$ into P_A^l. The reason is the following: by Theorem 3.1, for any fixed integer $A \in I_N$ there is an N-partition of $\mathbf{P}$ into P_A^l. The claim follows since there are N distinct values of A. As will be discussed subsequently, by choosing the values of A appropriately, Theorem 3.1 can be used to show that N-partitions of image $\mathbf{P}$ into subimages, meshes, rows, as well as columns as defined in Chapter 2 are all possible. The consequence of N-partitions thus obtained is that any subimage, mesh, row, or column is parallel accessible. In what follows, superscripts are used as repetition factors when describing binary numbers, e.g., $0^2 1^2 = 0011$.

Corollary 3.1: Any subimage of image $\mathbf{P}$ is parallel accessible.

Proof: It is sufficient to show that there exists an N-partition of $\mathbf{P}$ into P_A^l such that each P_A^l corresponds to a subimage of image $\mathbf{P}$.

By Theorem 3.1, $\Pi = \{\mathbf{P}_A^l \mid l \in \mathbf{I}_N\}$ is an N-partition of $\mathbf{P}$, where $A \in \mathbf{I}_N$. The following is to show that if integer A satisfies the condition $A = [0^{\frac{n}{2}} 1^{\frac{n}{2}}]$, then $\mathbf{P}_A^l$ corresponds to subimage $\xi(l)$.

Let integer A satisfy this condition, then pixel $p_{u,v}$ belongs to $\mathbf{P}_A^l$, where $u = \xi(l \oplus [1^{\frac{n}{2}} 0^{\frac{n}{2}}]k)$, $v = l \oplus [0^{\frac{n}{2}} 1^{\frac{n}{2}}]k$, and $k \in \mathbf{I}_N$ by the definition of $\mathbf{P}_A^l$.

Manipulating u and v as follows

$$\begin{aligned} u &= \xi(l \oplus [1^{\frac{n}{2}} 0^{\frac{n}{2}}]k) \\ &= \xi(l) \oplus \xi([1^{\frac{n}{2}} 0^{\frac{n}{2}}]k) \\ &= \xi(l) \oplus [0^{\frac{n}{2}} 1^{\frac{n}{2}}]\xi(k) \\ &= r \times \Gamma(\xi(l)) + \Gamma(l) \oplus \Gamma(k), \end{aligned}$$

and

$$\begin{aligned} v &= l \oplus ([0^{\frac{n}{2}} 1^{\frac{n}{2}}]k) \\ &= r \times \Omega(\xi(l)) + \Omega(l) \oplus \Omega(k). \end{aligned}$$

Recall the definition of a subimage and observe that both $\Gamma(l) \oplus \Gamma(k)$ and $\Omega(l) \oplus \Omega(k)$ are in $\mathbf{I}_r$. It is then not difficult to see that the $\mathbf{P}_A^l$ corresponds to the $\xi(l)$th subimage of image $\mathbf{P}$.

Q.E.D.

The parallel accessible property of the meshes of $\mathbf{P}$ can be established in a similar way as follows.

Corollary 3.2: Any mesh of image $\mathbf{P}$ is parallel accessible.

Proof: By Theorem 3.1, $\Pi = \{\mathbf{P}_A^l \mid l \in \mathbf{I}_N\}$ is an N-partition of $\mathbf{P}$, where $A \in \mathbf{I}_N$. The following is to show that if the integer A satisfies the condition $A = [1^{\frac{n}{2}} 0^{\frac{n}{2}}]$, then $\mathbf{P}_A^l$ corresponds to mesh l of $\mathbf{P}$.

Now, let A satisfy this condition, then pixel $p_{u,v}$ belongs to $\mathbf{P}'_A$, where $u = \xi(l \oplus [0^{\frac{n}{2}} 1^{\frac{n}{2}}]k)$, $v = l \oplus [1^{\frac{n}{2}} 0^{\frac{n}{2}}]k$, and $k \in \mathbf{I}_N$ by definition.

Manipulating the two expressions of u and v as follows

$$\begin{aligned} u &= \xi(l \oplus [0^{\frac{n}{2}} 1^{\frac{n}{2}}]k) \\ &= \xi(l) \oplus \xi([0^{\frac{n}{2}} 1^{\frac{n}{2}}]k) \\ &= \xi(l) \oplus [1^{\frac{n}{2}} 0^{\frac{n}{2}}]\xi(k) \\ &= \Gamma(l) + r \times (\Omega(l) \oplus \Omega(k)), \end{aligned}$$

and

$$\begin{aligned} v &= l \oplus ([1^{\frac{n}{2}} 0^{\frac{n}{2}}]k) \\ &= \Omega(l) + r \times (\Gamma(l) \oplus \Gamma(k)). \end{aligned}$$

Recall the definition of the meshes of image $\mathbf{P}$, and observe that both $\Omega(l) \oplus \Omega(k)$ and $\Gamma(l) \oplus \Gamma(k)$ are in $\mathbf{I}_r$. It is then easy to see that the $\mathbf{P}'_A$ corresponds to the lth mesh of image $\mathbf{P}$.

Q.E.D.

By choosing the value of integer A, Theorem 3.1 can be used to show that there is an N-partition of $\mathbf{P}$ into rows. Consequently, any row of the image is parallel accessible.

Corollary 3.3: Any row of image $\mathbf{P}$ is parallel accessible.

Proof: It is sufficient to show that there is an N-partition of $\mathbf{P}$ into $\mathbf{P}'_A$ such that each $\mathbf{P}'_A$ corresponds to a row of image $\mathbf{P}$.

Let the integer A satisfies the condition $A = [1^n]$, then there exists an N-partition of $\mathbf{P}$ into $\mathbf{P}'_A = \{p_{\xi(l), l \oplus k} \mid k \in \mathbf{I}_N\}$ due to Theorem 3.1.

Recall the definition of rows and observe that $l \oplus k \in \mathbf{I}_N$. This reveals that $\mathbf{P}'_A$ corresponds exactly to the $\xi(l)$th row of image $\mathbf{P}$.

Q.E.D.

Similarly, the parallel accessible property of columns of image $\mathbf{P}$ can be established.

Corollary 3.4: Any column of image $\mathbf{P}$ is parallel accessible.

Proof: The following shows that there exists an partition of $\mathbf{P}$ into $\mathbf{P}_A^l$ such that each $\mathbf{P}_A^l$ corresponds to a column of image $\mathbf{P}$.

If the integer A satisfies the condition $A = [0^n]$, then there is an N-partition of $\mathbf{P}$ into $\mathbf{P}_A^l = \{p_{\xi(l\oplus k),l} \mid k \in \mathbf{I}_N\}$ by Theorem 3.1.

Recall the definition of columns of image $\mathbf{P}$ and observe that $\xi(l\oplus k) \in \mathbf{I}_N$. This indicates that $\mathbf{P}_A^l$ corresponds to the lth column of image $\mathbf{P}$.

Q.E.D.

It is instructive to verify the above four corollaries using the memory module matrix of Fig. 3.2 for the case of $N = 16$ to gain some intuition about set $\mathbf{P}_A^l$. The parallel accessible property of $\mathbf{P}_A^l$ is a very important one, because conflict-free memory access is guaranteed as far as accessing the N elements of $\mathbf{P}_A^l$ in $\mathbf{M}$ is concerned. It is, however, also important to have an efficient algorithm capable of generating an address for each memory module of $\mathbf{M}$ simultaneously so that all elements of $\mathbf{P}_A^l$ can be located in parallel.

As it turns out, there is a simple and efficient algorithm for generating an address for each memory module in parallel. Consider the problem of locating pixel $p_{\xi(l\oplus Ak),l\oplus Ak}$ of $\mathbf{P}_A^l$ in $\mathbf{M}$. It resides in $\mathbf{M}_{k,l\oplus Ak}$ according to the storage scheme $\mathbf{S}$. Therefore, it is sufficient to evaluate the expression $l\oplus Ak$, giving the memory address for $\mathbf{M}_k$ to locate the pixel under consideration. Clearly, this expression is computable in constant time in terms of the number of gate delays. It is then necessary to use N half adders to construct an address generation mechanism to

generate the N addresses for the N memory modules in parallel. This not only results in a considerable saving in hardware but also in computation time.

As mentioned earlier, the storage scheme **S** does not preserve the logic order of the elements of image **P** stored in **M**. The rearranging or alignment of the elements of $\mathbf{P}_A^l$ becomes necessary in order to restore their order before they can be processed by the PE's. The problem of alignment of $\mathbf{P}_A^l$ will be discussed shortly.

Before preceding any further, the next section is devoted to describe the topological configuration of the N PE's with which the logic orderings of the PE's can be specified.

3.4. The PE's topology T

As depicted in Fig. 3.1 there are N identical PE's in the multiple-processor architecture. They recieve the same instruction broadcast from the CUN, and execute it on their own data in parallel. While each PE has its own local memory to store temporary results, the N PE's can access the memory system **M**, through the interconnection network, to retrieve or modify the N elements of some $\mathbf{P}_A^l$ simultaneously.

The topological configuration of the PE's is specified as follows. The N PE's are arranged logically as a two-dimensional $r \times r$ array, and indexed from 0 to $N-1$ in row-major order as shown in Fig. 3.3 for the case of $N = 16$. The PE at the ith row and the jth column of the array is also called PE (i,j), for $i,j \in I_r$. Clearly, PE_k and PE $(\Gamma(k),\Omega(k))$ refer to the same PE.

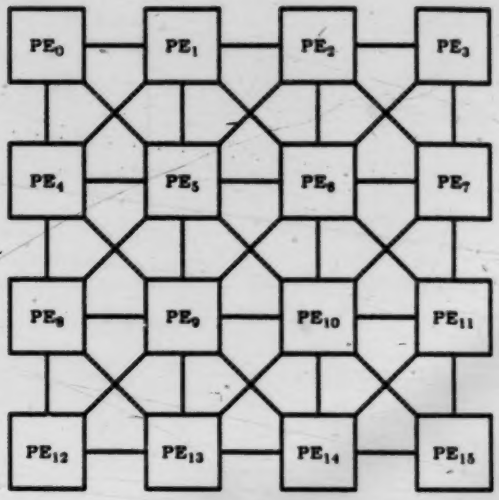

Fig. 3.3. An array of PE's for $N = 16$.

Each PE has eight immediate neighbors except those located at the edges of the array, and is capable of accessing one of its neighbors at any instant directly through the nearest neighbor connection in a synchronized manner. The eight neighbors of a PE will be numbered from 0 to 7 as follows:

$$\begin{array}{ccc} (\mathbf{PE}_k)_3 & (\mathbf{PE}_k)_4 & (\mathbf{PE}_k)_5 \\ (\mathbf{PE}_k)_2 & \mathbf{PE}_k & (\mathbf{PE}_k)_6 \\ (\mathbf{PE}_k)_1 & (\mathbf{PE}_k)_0 & (\mathbf{PE}_k)_7 \end{array}$$

where $(\mathbf{PE}_k)_i$ denotes the ith neighbor of $\mathbf{PE}_k$, for $0 \leq i \leq 7$. Thus with the eight neighbor connections, PE k for all $k \in \mathbf{I}_N$ can access PE $F_i(k)$ simultaneously, where $F_i(k)$ is the index of the PE that is the ith neighbor of $\mathbf{PE}_k$, provided such a neighbor exists.

The reason for the use of 8-connected mesh connections is that it allows the

utilization of neighborhood parallelism in the sense that with one shift instruction to be described later each processing element can obtain a pixel from any of its eight neighbors. An alternative topological configuration of the PE's can be 4-connected mesh connections. Then, each processing element has to spend two shift instructions to obtain a pixel from any of its four corner-neighbors.

A simplified architecture of the processing element is schematically shown in Fig. 3.4. There are a set of four general-purpose registers, a mask register, an arithmetic logic unit (ALU), a local memory (LOCM), and a multiplexer (MUX).

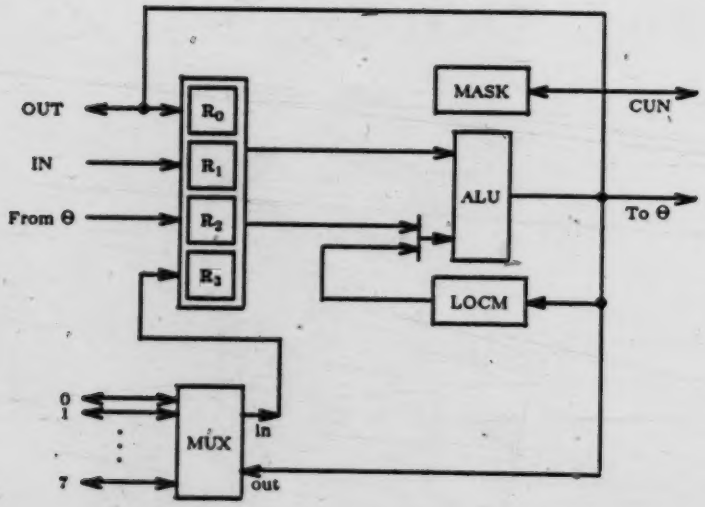

Fig. 3.4. A simplified architecture of PE.

The PE can read data from the memory system M into any of the four registers by connecting the "From Θ" line, which is an output terminal of the interconnection network, to the register. On the other hand, The PE can write the contents of any of these registers into the memory system M by connecting

the register through ALU to the "to Θ" line, which is an input terminal of the interconnection network. The multiplexer is used to interconnect the PE and its eight neighbors, where the link with label i, $0 \leq i \leq 7$, is used to interconnect PE_k and its ith neighbor, i.e., PE $F_i(k)$. Thus, connecting R_0 (through ALU) and R_1 to the multiplexer's "out" and "in" terminals, respectively, will allow PE k to load its own R_1 register with the contents of R_0 register of PE $F_i(k)$, for some $0 \leq i \leq 7$ and for all $k \in I_N$ in parallel. The mask register contains the index of the PE. A match between the contents of the mask register and the mask field of the instruction broadcast by the CUN through the "CUN" line will enable the PE to obey the instruction. Otherwise, the PE does nothing. The masking scheme can be extented to also permit masking to be set by a local condition. When this is the case, the mask register should contain extra bits as the processing element flags which can be set by the output of the ALU. Then, a processing element will have the option of executing an instruction or not, depending on the state of one of the processing element flags. The "IN" and "OUT" lines are for the purpose of image input and output which will be elaborated.

The use of the N processing elements with the described architecture will be further discussed in subsequent chapters. The following sections address issues regarding the movement of $\mathbf{P}_A^l$ between PE's and $\mathbf{M}$

3.5. The interconnection network Θ

The necessity of interposing an interconnection network between the memory system $\mathbf{M}$ and the N PE's is primarily due to the data alignment requirements of $\mathbf{P}_A^l$ imposed by the storage scheme $\mathbf{S}$. The data alignment requirements of $\mathbf{P}_A^l$ will be examined first, and a multistage interconnection network referred to as a theta network is then presented in this section which provides the capability for the necessary data alignments. Construction of this net-

work is described and many of its useful properties are given.

3.5.1. Data alignment requirements

Up to this point various N-partitions of $\mathbf{P}$ into $\mathbf{P}_A^l$ have been described. It is timely to discuss how they can be used for image processing, and what data alignment requirements are needed. The problem of alignment of $\mathbf{P}_A^l$ can be stated as follows. Given a $\mathbf{P}_A^l$, to take the N elements of $\mathbf{P}_A^l$ out of the memory system $\mathbf{M}$, and place them in some specified order in the N PE's for subsequent processing is a requirement. After they are processed by the N PE's it may also be necessary to change the ordering in order to write the results in $\mathbf{M}$ appropriately, i.e., place the resulting N elements in $\mathbf{M}$ in some specified order. Alignment of $\mathbf{P}_A^l$ here deliberatively means to place the N elements of $\mathbf{P}_A^l$ from $\mathbf{M}$ on the N PE's in some specified order and vice versa. A few examples illustrating the alignment requirements for subimage, mesh, row, and column of image $\mathbf{P}$ described previously should be helpful at this point in time.

Consider subimage 1 of image $\mathbf{P}$ for the case of $N = 16$. The subimage is displayed below:

$$\begin{matrix} p_{0,4} & p_{0,5} & p_{0,6} & p_{0,7} \\ p_{1,4} & p_{1,5} & p_{1,6} & p_{1,7} \\ p_{2,4} & p_{2,5} & p_{2,6} & p_{2,7} \\ p_{3,4} & p_{3,5} & p_{3,6} & p_{3,7} \end{matrix}$$

As can be seen, the 16 elements have logic order in the sense that each one has well-defined neighbors. For instance, $p_{1,5}$ has eight immediate neighbors surrounding it. Recall that the 16 PE's also have logic order when arranged as a 4×4 array as shown in Fig. 3.3. Each PE has well-defined neighbors in the array. It is desirable to place the subimage on the PE array in such a way that neighboring picture elements are placed in neighboring PE's. That is, place $p_{1,5}$ in PE_5, and place the neighboring pixels of $p_{1,5}$ in the corresponding neighboring PE's of

PE_5, and so on. The value of such an arrangement lies in its use for effective supporting local neighborhood operation algorithms.

Fig. 3.5 illustrates the 16 pixels of the subimage and the corresponding memory modules in which they are actually stored, as indicated by the underscores in the memory module matrix.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3
1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10
9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6
13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2
2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13
6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9
10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5
14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1
3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12
7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fig. 3.5. Subimage distribution.

It is apparent that in order to achieve the desired arrangement the memory modules must be connected to the PE's in such a way that memory module 0 which contains $p_{1,4}$ is connected to PE_4, memory module 1 which contains $p_{1,5}$ is connected to PE_5, and so on. More precisely, the alignment of the subimage requires the following mapping of memory modules onto PE's:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 \end{pmatrix},$$

where a mapping π is displayed in explicit form by placing $\pi(k)$ directly below k in the 2×16 array to mean that M_k is connected to $PE_{\pi(k)}$. With such a memory-processor connection configuration, the elements of the subimage can be placed on the PE array in the desired order.

Consider the alignment requirement of mesh 1 of image **P** for the case of $N = 16$. The mesh is shown below:

$$\begin{matrix} p_{0,1} & p_{0,5} & p_{0,9} & p_{0,13} \\ p_{4,1} & p_{4,5} & p_{4,9} & p_{4,13} \\ p_{8,1} & p_{8,5} & p_{8,9} & p_{8,13} \\ p_{12,1} & p_{12,5} & p_{12,9} & p_{12,13} \end{matrix}$$

The 16 elements of the mesh belong to distinct subimages of which each has neighboring subimages. It is important to observe that the 16 elements have logic order in the above sense. For instance, $p_{4,5}$ has eight neighbors surrounding it, which belong to the eight neighboring subimages of the subimage to which $p_{4,5}$ belongs.

It is desirable to place the mesh on the PE array in such a way that neighboring pixels are placed in neighboring PE's. That is, place $p_{4,5}$ in PE_5, and place the neighboring pixels of $p_{4,5}$ in the corresponding neighboring PE's of PE_5, and so on. The value of such an arrangement lies in its use for supporting global neighborhood operation algorithms effectively.

Fig. 3.6 illustrates the 16 pixels of the mesh and the corresponding memory modules in which they are actually stored, as indicated by the underscores in the memory module matrix.

$$\begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 \\
8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\
1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 & 9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 \\
5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 & 13 & 12 & 15 & 14 & 9 & 8 & 11 & 10 \\
9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 & 1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 \\
13 & 12 & 15 & 14 & 9 & 8 & 11 & 10 & 5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 \\
2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 & 10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 \\
6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 & 14 & 15 & 12 & 13 & 10 & 11 & 8 & 9 \\
10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 & 2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 \\
14 & 15 & 12 & 13 & 10 & 11 & 8 & 9 & 6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 \\
3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 & 11 & 10 & 9 & 8 & 15 & 14 & 13 & 12 \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 \\
11 & 10 & 9 & 8 & 15 & 14 & 13 & 12 & 3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 \\
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{pmatrix}$$

Fig. 3.6. Mesh distribution.

It is easy to see that in order to achieve the desired arrangement the memory modules must be connected to the PE's in such a way that memory module 0 which contains $p_{4,1}$ is connected to PE_4, memory module 1 which contains $p_{0,1}$ is connected to PE_0, and so on. More precisely, the following mapping of memory modules onto PE's is required:

$$\begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
4 & 0 & 12 & 8 & 5 & 1 & 13 & 9 & 6 & 2 & 14 & 10 & 7 & 3 & 15 & 11
\end{pmatrix}.$$

With such a memory-processor connection configuration, the elements of the mesh can be placed on the PE array in the desired order. The alignment requirement is, however, different from the previous example.

Now, consider the alignment requirement of the last row of image **P** for the case of $N = 16$. The row is displayed below:

$$p_{15,0}\ p_{15,1}\ \cdots\ p_{15,15}.$$

A desirable placement of the row on PE array is such that the column index of the pixel agrees with the index of the PE's in which they are stored. That is, place the kth pixel of the row in PE_k. For example, place $p_{15,0}$ in PE_0, and place $p_{15,1}$ in PE_1, and so on. The value of such an arrangement lies in its use for efficient implementation of parallel algorithms such as matrix manipulations, as well as image input/output operations.

Fig. 3.7 illustrates the 16 pixels of the row and the memory modules in which they are stored, as indicated by the underscores in the memory module matrix.

$$\begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 \\
8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\
1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 & 9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 \\
5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 & 13 & 12 & 15 & 14 & 9 & 8 & 11 & 10 \\
9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 & 1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 \\
13 & 12 & 15 & 14 & 9 & 8 & 11 & 10 & 5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 \\
2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 & 10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 \\
6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 & 14 & 15 & 12 & 13 & 10 & 11 & 8 & 9 \\
10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 & 2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 \\
14 & 15 & 12 & 13 & 10 & 11 & 8 & 9 & 6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 \\
3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 & 11 & 10 & 9 & 8 & 15 & 14 & 13 & 12 \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 \\
11 & 10 & 9 & 8 & 15 & 14 & 13 & 12 & 3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 \\
\underline{15} & \underline{14} & \underline{13} & \underline{12} & \underline{11} & \underline{10} & \underline{9} & \underline{8} & \underline{7} & \underline{6} & \underline{5} & \underline{4} & \underline{3} & \underline{2} & \underline{1} & \underline{0}
\end{pmatrix}$$

Fig. 3.7. Row distribution.

It is apparent that the following mapping of memory modules onto PE's are required in order to place the pixels of the row on the PE array in the desired order.

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{pmatrix}.$$

Finally, consider the alignment requirement of column 15 of **P** for the case of $N = 16$. The column is displayed below:

$$p_{0,15} \quad p_{1,15} \quad \cdots \quad p_{15,15}.$$

It is desirable to place the kth pixel of the column in PE_k. For instance, place $p_{0,15}$ in PE_0, and place $p_{1,15}$ in PE_1, and so on. The reason for such an arrangement is similar to that for rows described earlier.

Again, Fig. 3.8 illustrates the 16 elements of the column and the memory modules in which they are stored, as indicated by the underscores in the memory module matrix.

$$\begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 \\
8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\
1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 & 9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 \\
5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 & 13 & 12 & 15 & 14 & 9 & 8 & 11 & 10 \\
9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 & 1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 \\
13 & 12 & 15 & 14 & 9 & 8 & 11 & 10 & 5 & 4 & 7 & 6 & 1 & 0 & 3 & 2 \\
2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 & 10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 \\
6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 & 14 & 15 & 12 & 13 & 10 & 11 & 8 & 9 \\
10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 & 2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 \\
14 & 15 & 12 & 13 & 10 & 11 & 8 & 9 & 6 & 7 & 4 & 5 & 2 & 3 & 0 & 1 \\
3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 & 11 & 10 & 9 & 8 & 15 & 14 & 13 & 12 \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 \\
11 & 10 & 9 & 8 & 15 & 14 & 13 & 12 & 3 & 2 & 1 & 0 & 7 & 6 & 5 & 4 \\
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0
\end{pmatrix}$$

Fig. 3.8. Column distribution.

Clearly, the following mapping of memory modules onto PE's is required to achieve the desired ordering:

$$\begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
15 & 11 & 7 & 3 & 14 & 10 & 6 & 2 & 13 & 9 & 5 & 1 & 12 & 8 & 4 & 0
\end{pmatrix}.$$

The data alignment requirements for subimage, mesh, row, and column of image $\mathbf{P}$ have been demonstrated by the preceding examples for the case of $N = 16$, where a number of mappings of the memory modules onto the PE's has been identified. In general, as will be seen later, the alignment of $\mathbf{P}_A^l$ requires two classes of mappings. One has the form of $\pi(k) = k \oplus f$, while the other has the form of $\pi(k) = \xi(k) \oplus f$, where f is a function of l. To realize these two classes of

mappings is where the interconnection network becomes necessary.

3.5.2. Construction of a theta network

In this section, the interconnection network called a theta network is introduced to realize the alignment functions for P_A^l. The section begins with a description of the construction of the theta network. An example of a 16×16 theta network is shown in Fig. 3.9. The realizable mappings of the network inputs onto the network outputs is accomplished by two components: the interconnection links and the switching elements. The mappings of the network are also confined by its control structure referred to as routing vector. The routing vector of the network sets the states of each switching element so that data is allowed to flow from the input to the output.

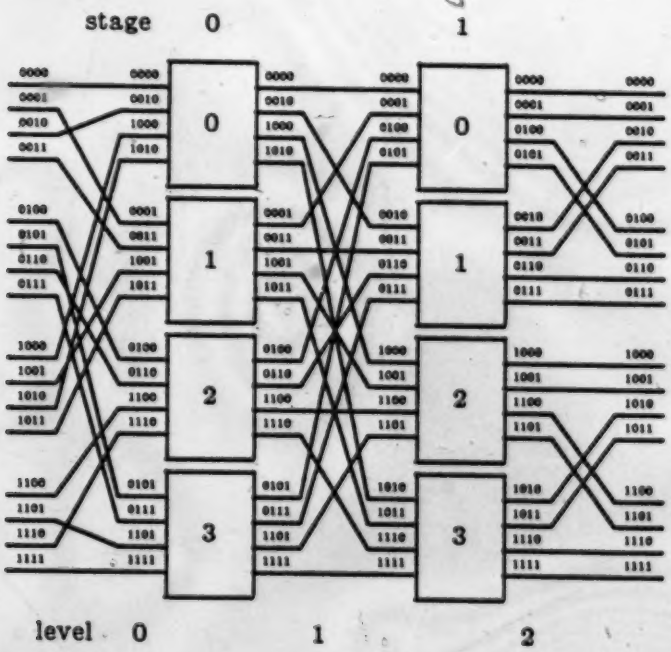

Fig. 3.9. The theta network for $N = 16$.

In general, an $N \times N$ theta network consists of $\frac{\log N}{2} = \frac{n}{2}$ stages and $\frac{n}{2}+1$ levels of interconnection links. Each stage consists of $\frac{N}{4}$ switching elements. Each switching element has four input terminals, four output terminals, and can be put into one of the eight states as shown in Fig. 3.10.

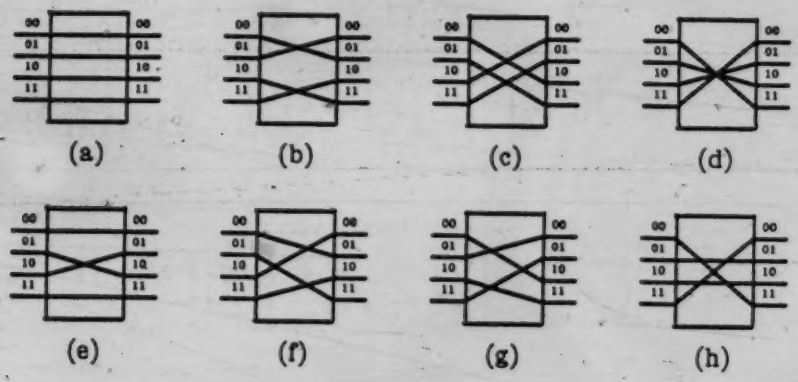

Fig. 3.10. Eight possible states of a switching element.

To describe the switching element more precisely, let the input terminals and the output terminals be indexed from 0 to 3, respectively, as indicated in Fig. 3.10. Let α and c, where $0 \leq \alpha \leq 1$ and $0 \leq c \leq 3$, together be the control variable that sets the switching element to one of the above eight states. Then, input terminal i, for $0 \leq i \leq 3$, of the switching element can be switched to output terminal $\sigma_c^{\alpha}(i)$ defined as

$$\sigma_c^{\alpha}(i) = \begin{cases} i \oplus c & \text{if } \alpha = 0 \\ \xi(i) \oplus c & \text{if } \alpha = 1 \end{cases}$$

That is, a switching element can perform the following eight one-to-one and onto mappings of its input terminals to its output terminals:

$$\sigma_0^0 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{pmatrix}, \sigma_1^0 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \end{pmatrix}, \sigma_2^0 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 3 & 0 & 1 \end{pmatrix}, \sigma_3^0 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \end{pmatrix},$$

and

$$\sigma_0^1 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 2 & 1 & 3 \end{pmatrix}, \sigma_1^1 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \end{pmatrix}, \sigma_2^1 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 3 & 1 \end{pmatrix}, \sigma_3^1 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 1 & 2 & 0 \end{pmatrix},$$

where a mapping σ_c^a is displayed in explicit form by placing $\sigma_c^a(i)$ directly below i in the 2×4 array to mean that the ith input terminal of the switching element is connected to the $\sigma_c^a(i)$-th output terminal.

The stages of the network are labeled in a sequence from 0 to $\frac{n}{2}-1$ with 0 for the leftmost stage and $\frac{n}{2}-1$ for the rightmost stage. Similarly, the levels of the links are labeled in a sequence from 0 to $\frac{n}{2}$. Switching elements at each stage are labeled in a sequence from 0 to $\frac{N}{4}-1$ with 0 for the top switching element and $\frac{N}{4}-1$ for the bottom one.

Let each switching element in each stage use $q_0q_1 \cdots q_{u-1}$ as a binary representation of its label, where $u = \log\frac{N}{4} = n-2$.

Define λ_j^i to be an integer function

$$\lambda_j^i(q_0q_1 \cdots q_{u-1}) = q_0 \cdots q_{j-1}i_0q_j \cdots q_{\frac{n}{2}+j-1}i_1q_{\frac{n}{2}+j} \cdots q_{u-1},$$

where $0 \le i \le 3$, $0 \le j \le u-1$, and $i_0=\lfloor\frac{i}{2}\rfloor$, $i_1= i \bmod 2$.

The function λ_j^i is calculated by simply inserting i_0 and i_1 (the most significant bit and the least significant bit of integer i) into the front of the jth bit and the front of $(\frac{n}{2}+j)$-th bit of the $(n-2)$-bit binary number $q_0q_1 \cdots q_{u-1}$, respectively, resulting in an n-bit binary number. With the aid of this function, the N input terminals and N output terminals of each stage can be labeled according to the following scheme.

Label scheme: The ith input terminal and the ith output terminal of switching element $q_0q_1 \cdots q_{n-1}$ in stage j both have the same label with the value of $\lambda_j^i(q_0q_1 \cdots q_{n-1})$.

For example of the label scheme, see Fig. 3.9 for the case of $N = 16$. Next, the topology of the theta network that defines the connectivities of the links between switching elements in adjacent stages is now described by the topology scheme given below.

Topology scheme: Create a link to interconnect an output terminal of stage $j-1$ (or a network input for the case of level 0) and an input terminal of stage j (or a network output for the case of level $\frac{n}{2}$) if these two terminals have identical label. The same label is used as the label of the link created.

For example of this topology scheme, see Fig. 3.9 for the case of $N = 16$. In order to switch data through the network, each switching element in the network must be set to one of the above eight states and then data is allowed to pass from inputs to outputs.

Two questions arise at this point. First, what kinds of one-to-one mappings of inputs into outputs can be generated by this network in one pass with simple control? Second, how can the necessary states for the switching elements be determined for possible mappings? These two questions will be discussed in the following sections.

3.5.3. Properties of the network

This section will demonstrate two classes of one-to-one mappings of the network inputs into the network outputs which can be produced by the theta network in one pass. It will be shown that armed with these two classes of mappings, the theta network is able to align various types of $\mathbf{P}_A^l$.

Definition: Define $\mathbf{A}_N$ to be a set of integer pairs $\mathbf{A}_N = \{(I_i, O_i) \mid i \in \mathbf{I}_N\}$ which represents an one-to-one and onto mapping of the network inputs to the network outputs:

$$\begin{pmatrix} I_0 & I_1 & \cdots & I_i & \cdots & I_{N-1} \\ O_0 & O_1 & \cdots & O_i & \cdots & O_{N-1} \end{pmatrix}.$$

Definition: Let Θ be the $N \times N$ theta network as described before. If Θ can generate the mapping specified by $\mathbf{A}_N$ then Θ is said to realize $\mathbf{A}_N$. Otherwise, Θ does not realize $\mathbf{A}_N$.

In what follows, symbols l_i and k_i are used to denote the ith bit of the binary representation of integers l and k, respectively. Also, α and c are used as the control variable of the switching elements as before.

Theorem 3.3: Let $\mathbf{A}_N = \{(k, k \oplus l) \mid k \in \mathbf{I}_N\}$ where $l \in \mathbf{I}_N$. Θ realizes $\mathbf{A}_N$.

Proof: Let the control variables of all the switching elements in stage i have identical values: $\alpha = 0$ and $c = l_i l_{\frac{n}{2}+i}$, for $0 \le i \le \frac{n}{2}-1$.

The induction hypothesis (to be proven correct) is that after passing stage i network input k will be currently connected to the link of level $i+1$ whose label is $k_0 \oplus l_0 \cdots k_i \oplus l_i \, k_{i+1} \cdots k_{\frac{n}{2}-1} \, k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \cdots k_{\frac{n}{2}+i} \oplus l_{\frac{n}{2}+i} \, k_{\frac{n}{2}+i+1} \cdots k_{n-1}$ in binary representation. Thus, when $i = \frac{n}{2}-1$, network input k will be connected to network output $k_0 \oplus l_0 \cdots k_{\frac{n}{2}-1} \oplus l_{\frac{n}{2}-1} \, k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \, k_{n-1} \oplus l_{n-1}$ which is $k \oplus l$. To complete the proof it must be

shown that the induction hypothesis is true.

Basis: $i = 0$.

According to the label scheme and topology scheme, network input $k = k_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}$ is connected to the $(k_0 k_{\frac{n}{2}})$-th input terminal of switching element $k_1 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}+1} \cdots k_{n-1}$ at stage 0. Under the control variable $\alpha = 0$ and $c = l_0 l_{\frac{n}{2}}$ assumed, this input terminal is switched to output terminal $\sigma^0_{l_0 l_{\frac{n}{2}}}(k_0 k_{\frac{n}{2}}) = k_0 \oplus l_0 k_{\frac{n}{2}} \oplus l_{\frac{n}{2}}$ of the same switching element. The label of this output terminal is, by the label scheme, $k_0 \oplus l_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \cdots k_{n-1}$. Thus, by the topology scheme, network input $k = k_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}$ is connected to link $k_0 \oplus l_0 k_1 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} k_{\frac{n}{2}+1} \cdots k_{n-1}$ of level 1.

Induction step: Assume true for $i = j-1$ and show true for $i = j$.

From the induction hypothesis when $i = j-1$, network input $k = k_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}$ is connected to the link of level j whose label is $k_0 \oplus l_0 \cdots k_{j-1} \oplus l_{j-1} k_j \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \cdots k_{\frac{n}{2}+j-1} \oplus l_{\frac{n}{2}+j-1} k_{\frac{n}{2}+j} \cdots k_{n-1}$. This link is, however, connected to the $(k_j k_{\frac{n}{2}+j})$-th input terminal of switching element $k_0 \oplus l_0 \cdots k_{j-1} \oplus l_{j-1} k_{j+1} \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \cdots k_{\frac{n}{2}+j-1} \oplus l_{\frac{n}{2}+j-1} k_{\frac{n}{2}+j+1} \cdots k_{n-1}$ in stage j. Under the control variable $\alpha = 0$ and $c = l_j l_{\frac{n}{2}+j}$, this input terminal is switched to output terminal $\sigma^0_{l_j l_{\frac{n}{2}+j}}(k_j k_{\frac{n}{2}+j}) = k_j \oplus l_j k_{\frac{n}{2}+j} \oplus l_{\frac{n}{2}+j}$ of the same switching element. According to the label scheme, the label of this output terminal is $k_0 \oplus l_0 \cdots k_j \oplus l_j k_{j+1} \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \cdots k_{\frac{n}{2}+j} \oplus l_{\frac{n}{2}+j} \quad k_{\frac{n}{2}+j+1} \cdots k_{n-1}$. According to the topology scheme, network input $k = k_0 \cdots k_{n-1}$ is currently

connected to link $k_0 \oplus l_0 \cdots k_j \oplus l_j k_{j+1} \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} \cdots k_{\frac{n}{2}+j} \oplus l_{\frac{n}{2}+j} k_{\frac{n}{2}+j+1} \cdots k_{n-1}$ of level $j+1$. Thus, the induction proof is true for $i = j$. This completes the proof.

Q.E.D.

Theorem 3.4: Let $A_N = \{(k, \xi(k) \oplus l) \mid k \in I_N\}$ where $l \in I_N$. Θ realizes A_N.

Proof: Let control variables of all switching elements in stage i have identical values: $\alpha = 1$ and $c = l_i l_{\frac{n}{2}+i}$.

The induction hypothesis (to be proven correct) is that after passing stage i network input k will be currently connected to the link of level $i+1$ whose label is $k_{\frac{n}{2}} \oplus l_0 \cdots k_{\frac{n}{2}+i} \oplus l_i k_{i+1} \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} \cdots k_i \oplus l_{\frac{n}{2}+i} k_{\frac{n}{2}+i+1} \cdots k_{n-1}$.

Thus, when $i = \frac{n}{2} - 1$, network input k will be connected to network output $k_{\frac{n}{2}} \oplus l_0 \cdots k_{n-1} \oplus l_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} \cdots k_{\frac{n}{2}-1} \oplus l_{n-1}$ which is $\xi(k) \oplus l$. To complete the proof it must be shown that the induction hypothesis is true.

Basis: $i = 0$.

By the label scheme and topology scheme, network input $k = k_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}$ is connected to the $(k_0 k_{\frac{n}{2}})$-th input terminal of switching element $k_1 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}+1} \cdots k_{n-1}$ in stage 0. Under the control variable $\alpha = 1$ and $c = l_0 l_{\frac{n}{2}}$ assumed, this input terminal is switched to output terminal $\sigma^1_{l_0 l_{\frac{n}{2}}}(k_0 k_{\frac{n}{2}}) = k_{\frac{n}{2}} \oplus l_{\frac{n}{2}} k_0 \oplus l_0$ of the same switching element. Recall the label scheme, this output terminal's label is $k_{\frac{n}{2}} \oplus l_0 k_1 \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} k_{\frac{n}{2}+1} \cdots k_{n-1}$. Thus, by the topology scheme, network input $k = k_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}$ is

connected to link $k_{\frac{n}{2}} \oplus l_0 k_1 \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} k_{\frac{n}{2}+1} \cdots k_{n-1}$ of level 1.

Induction step: Assume true for $i = j-1$ and show true for $i = j$.

From the induction hypothesis when $i = j-1$, the network input $k = k_0 \cdots k_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}$ is connected to the link of level j whose label is $k_{\frac{n}{2}} \oplus l_0 \cdots k_{\frac{n}{2}+j-1} \oplus l_{j-1} k_j \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} \cdots k_{j-1} \oplus l_{\frac{n}{2}+j-1} k_{\frac{n}{2}+j} \cdots k_{n-1}$. This link is, however, connected to the $(k_j k_{\frac{n}{2}+j})$-th input terminal of switching element $k_{\frac{n}{2}} \oplus l_0 \cdots k_{\frac{n}{2}+j-1} \oplus l_{j-1} k_{j+1} \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} \cdots k_{j-1} \oplus l_{\frac{n}{2}+j-1} k_{\frac{n}{2}+j+1} \cdots k_{n-1}$ in stage j. With the control variable $\alpha = 1$ and $c = l_j l_{\frac{n}{2}+j}$, this input terminal is switched to output terminal $\sigma^1_{l_j l_{\frac{n}{2}+j}}(k_j k_{\frac{n}{2}+j}) = k_{\frac{n}{2}+j} \oplus l_j k_j \oplus l_{\frac{n}{2}+j}$ of the same switching element. According to the label scheme, the label of this output terminal is $k_{\frac{n}{2}} \oplus l_0 \cdots k_{\frac{n}{2}+j} \oplus l_j k_{j+1} \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} \cdots k_j \oplus l_{\frac{n}{2}+j} \quad k_{\frac{n}{2}+j+1} \cdots k_{n-1}$. According to the topology scheme, network input $k = k_0 \cdots k_{n-1}$ is currently connected to link $k_{\frac{n}{2}} \oplus l_0 \cdots k_{\frac{n}{2}+j} \oplus l_j k_{j+1} \cdots k_{\frac{n}{2}-1} k_0 \oplus l_{\frac{n}{2}} \cdots k_j \oplus l_{\frac{n}{2}+j} k_{\frac{n}{2}+j+1} \cdots k_{n-1}$ of level $j+1$. Thus, the induction proof is true for $i = j$. This completes the proof.

Q.E.D.

The proofs of these two theorems convey useful information regarding the control of the theta network in generating the required mappings. The necessary states for the switching elements can be determined easily. First, all switching elements in the same stage are set to identical states. Second, the routing vector is composed of two parts, i.e., α and l. The first part is only one bit and applied to all stages uniformly, while the second part is of n bits of which the ith bit and the $(i+\frac{n}{2})$th bit are only applied to the ith stage of the network. In other

words, the routing vector has $\frac{n}{2}$ components of 3 bits each, where the ith component consists of α, l_i and $l_{\frac{n}{2}+i}$ and is used to set the state of all switching elements in stage i. The construction of the routing vector of the Θ network in order to align the N elements of various $\mathbf{P}_A^l$ will be discussed shortly.

It is interesting to note that one of the two classes of mappings realizable by the theta network can not be realized by existing multistage networks such as Flip network [4], Omega network [23], and indirect binary n-cube network [31]. Although these well-known multistage interconnection networks can realize the class of permutations in Theorem 3.3 in one pass, they are, however, unable to realize the class of permutations in Theorem 3.4 regardless the number of passes to be performed. This class of permutations is vitally important to the access and alignment of data in the proposed architecture. It is indeed the motivation for proposing the theta network.

As seen, the theta network presented is implemented with 4×4 switching elements, and consists of $\frac{n}{2}$ stages. It can be shown that the implementation of the theta network with 4×4 switching element is optimal for the set of permutations given with respect to the number of stages.

The theta network could be implemented with 2×2 switching elements also, resulting in a two-network implementation with $n+1$ stages in total as follows. The first stage forms one network that can realize either the mapping $A_N = \{ (k,k) \mid k \in \mathrm{I}_N \}$ or the mapping $A_N = \{ (k,\xi(k)) \mid k \in \mathrm{I}_N \}$, and the remaining n stages form another network which realizes the mapping $A_N = \{ (k,k \oplus l) \mid k \in \mathrm{I}_N \}$, where $l \in \mathrm{I}_N$. These two networks are cascaded in such a way that the kth output terminal of the first network is connected to the kth input terminal of the second network. The two-network implementation of the theta network can realize the two classes of permutations given as follows. To generate the one in Theorem

3.3, the first network is used to realize the mapping $\{(k,k)\}$, and the second network is used to realize the mapping $\{(k,k\oplus l)\}$. To generate the one in Theorem 3.4, the first network is used to realize the mapping $\{(k,\xi(k))\}$, and the second network is used to realize the mapping $\{(k,k\oplus l)\}$. It is apparent that the second network can be any of the three networks: Flip network, Omega network, and indirect binary n-cube network, with individual stage control.

The routing vector, in the two-network implementation, still consists of $n+1$ bits, where the α part is used to control the first network, while the remaining n bits are for the second network with l_i being used to control the ith stage of the second network. It is obvious that the two-network implementation operates more than twice as slower as the original implementation. In fact, the number of stages in the two-network implementation of the thete network can be further reduced to $\frac{n}{2}+1$, by using 4×4 switching elements with four possible states. Nevertheless, it is not possible to reduce it to $\frac{n}{2}$ stages.

The amount of bits used in the routing vector is $n+1$ for the two-network implementation, as opposed to $\frac{3}{2}n$ bits used in the routing vector for the original implementation.

3.6. Access and alignment of P_A^l

Armed with the Θ network, this section will consider two important operations: read and write, with which the PE's can access data in **M**. Notice that the multiple-processor architecture uses only one theta network, as depicted in Fig. 3.2, for both read and write operations. Since the theta network allows data flow in only one direction, the system has to use it differently according to the operation to be performed. For read operation, on the one hand, M_i is connected

to input terminal k, and PE_k is connected to output terminal k, allowing data to flow from M to PE's. For write operation, on the other hand, PE_k is connected to input terminal k, and M_k is connected to output terminal k to allow data to flow from PE's to M. Since read and write operations never occur simultaneously, such a system configuration is valid. It is apparent that the system could use two Θ networks, one to read data from M, and the other to write data into M, respectively. The use of two Θ networks should increase the operation speed, and at the same time increase the cost of the hardware in the system.

It is the responsibility of the CUN to establish these two different configurations accordingly for read and write operations. In addition, the CUN controls the address generating mechanism to locate N memory locations of M for either operation to be performed. Furthermore, the CUN controls the Θ network to route the data to the proper destination by means of the routing vector.

The task for the CUN to establish the above different configurations is rather trivial, and therefore will not be discussed here. The control of the address generating mechanism as well as the Θ network needs, however, further consideration. The following theorem concerns the read operation with which PE's read P_A^l.

Theorem 3.5: Either PE_k can read $p_{\xi(A^l \oplus \overline{A}k),\overline{A}l \oplus Ak}$ of P_A^l, for all $k \in I_N$; or PE_k can read $p_{\xi(A^l \oplus \overline{A}\xi(k)),\overline{A}l \oplus A\xi(k)}$ of P_A^l, for all $k \in I_N$.

Proof: Notice that M_k is connected to network input k, and PE_k is connected to network output k for read operation. Consider the following two cases.

Case 1: Let the routing vector of the Θ network be $0l$. By Theorem 3.3, the Θ network will generate the mapping of input k to output $k \oplus l$. Thus, $M_{k \oplus l}$ will be connected to PE_k through the interconnection network, allowing data to flow from $M_{k \oplus l}$ to PE_k. Furthermore, the address, which is generated by the address

generating mechanism from the two integers A and l, is $l \oplus A(k \oplus l)$ for $M_{k \oplus l}$. Therefore, data in $M_{k \oplus l, l \oplus A(k \oplus l)}$ will be fetched. According to the storage scheme **S**, data stored in that location is pixel $p_{\xi(l \oplus \overline{A}(k \oplus l)), l \oplus A(k \oplus l)}$, i.e., $p_{\xi(A l \oplus \overline{A} l), \overline{A} l \oplus A k}$.

Case 2: Let the routing vector of the Θ network be $1\xi(l)$. According to Theorem 3.4, the Θ network can generate the mapping of input k to output $\xi(k \oplus l)$. Hence, $M_{\xi(k) \oplus l}$ can be connected to PE_k, allowing data to flow from the memory module to the corresponding processing element. Furthermore, the address, which is generated by the address generating mechanism from the two integers A and l for $M_{\xi(k) \oplus l}$, is $l \oplus A(\xi(k) \oplus l) = \overline{A} l \oplus A \xi(k)$. Consequently, data in $M_{\xi(k) \oplus l, \overline{A} l \oplus A \xi(k)}$ will be fetched. According to the storage scheme **S**, the data stored in that location is pixel $p_{\xi(l \oplus \overline{A}(\xi(k) \oplus l)), \overline{A} l \oplus A \xi(k)}$, i.e., $p_{A l \oplus \overline{A} \xi(k), \overline{A} l \oplus A \xi(k)}$.

This completes the proof.

Q.E.D.

The constructive proof of Theorem 3.5 conveys some useful information about the interactions among **M**, the Θ network, PE's, and the CUN. A few comments on the role that the CUN plays should be helpful at this point.

Firstly, it is the responsibility of the CUN to provide the integers A and l to the address generating mechanism, which then generates addresses for all memory modules to locate $\mathbf{P}_A^l$ in **M** simultaneously. Secondly, the CUN controls the Θ network using either routing vector $0l$, or routing vector $1\xi(l)$. The N elements of $\mathbf{P}_A^l$ are then read out of **M**, and routed through the Θ network in parallel to the N PE's. The importance of placing $\mathbf{P}_A^l$ on the PE array in the orders specified by Theorem 3.5 will become clear shortly.

It is, however, interesting to know that after $\mathbf{P}_A^l$ has been routed from **M** to PE's by a read operation and processed there, how it or the updated version can be written from PE's back to where it was in **M** by a write operation?

Notice that the CUN will first connect PE_k and M_k to input terminal k and output terminal k, respectively, to allow data flow from PE's to **M**. The following two theorems concern the construction of the routing vector of the Θ network to route P_A^l from PE's back to **M** by a write operation.

Theorem 3.6: If P_A^l is routed from **M** to PE's through Θ under the control of routing vector αl by a read operation, then P_A^l can be routed back from PE's to where it was in **M** through Θ under the control of the same routing vector by a write operation.

Proof: For the read operation, the Θ network, under the control of the routing vector αl, allows data to flow from M_k to $PE_{k \oplus l}$, for all $k \in I_N$, by Theorem 3.3.

For the write operation, the Θ network, under the control of the same routing vector, allows the data at $PE_{k \oplus l}$ to flow to $M_{k \oplus l \oplus l}$ which is M_k. Moreover, let the address generating mechanism use integers A and l to generate the address for M_k. Then, the address generated is the same for both the write operation and the previous read operation, because the same integers A and l are used. Thus the data can be written back into the same memory location. The result then follows.

Q.E.D.

Theorem 3.6 indicates that as long as P_A^l is routed from **M** to PE's by a read operation using a routing vector whose α part has the value of 0, it can be routed back to **M** from PE's by a write operation without changing the routing vector. In addition, the same integers A and l are used by the address generating mechanism to locate N memory locations of **M** into which P_A^l will be written.

The next theorem considers the case where the α part of the routing vector has the value of 1.

Theorem 3.7: If P_A^l is routed from $\mathbf{M}$ through Θ to PE's under the control of the routing vector $1\xi(l)$ by a read operation, then P_A^l can be routed back from PE's to where it was in $\mathbf{M}$ through Θ under the control of the the routing vector $1l$ by a write operation.

Proof: For the read operation, the Θ network under the control of routing vector $1\xi(l)$ permits data to flow from M_k to $PE_{\xi(k\oplus l)}$, for all $k \in I_N$, by Theorem 3.4.

For the write operation, the Θ network under the control of the routing vector $1l$ allows the data at $PE_{\xi(k\oplus l)}$ to flow to $M_{\xi(\xi(k\oplus l))\oplus l}$, i.e., M_k. Moreover, let the address generating mechanism use integers A and l to produce the address for M_k. Then, the address generated for the write operation is the same as that for the previous read operation due to the same integers A and l used. Thus, the data can be written back to exactly where it was in $\mathbf{M}$, which completes the proof.

Q.E.D.

In contrast, Theorem 3.7 points out that if P_A^l is read by PE's from $\mathbf{M}$ with routing vector $1\xi(l)$, it can be written back to where it was in $\mathbf{M}$ by PE's but using a new routing vector $1l$. Again, the same integers A and l are used for address generation.

In conclusion, Theorems 3.6 and 3.7 together show that a bidirectional path between a memory module and a processing element can be established using only one Θ network. As a result, Theorem 3.5 which is concerned only with the read operation can be extended to include the write operation as well.

As shown by Corollaries 3.1, 3.2, 3.3, and 3.4, any submage, mesh, row, or column of image $\mathbf{P}$ is parallel accessible. The desired orderings of the pixels when placed on PE array have been illustrated by previous examples for the case of N

$= 16$. As an application of Theorems 3.5, 3.6, and 3.7, however, the following general results can be obtained.

The following corollary shows that the pixels of any subimage of $\mathbf{P}$ can be placed on the PE array in the desired order.

Corollary 3.5: For any given subimage of $\mathbf{P}$, PE (i,j) can access pixel (i,j) of that subimage, for all $i,j \in \mathbf{I}_r$.

Proof: From Corollary 3.1, $\mathbf{P}_A^l$ corresponds to subimage $\xi(l)$ when integer A satisfy the condition $A = [0^{\frac{n}{2}} 1^{\frac{n}{2}}]$. By Theorem 3.5, on the other hand, $\mathbf{PE}_k$, for all $k \in \mathbf{I}_N$, can read pixel $p_{u,v}$ from $\mathbf{M}$, where $u = \xi(Al \oplus \bar{A}k)$, and $v = \bar{A}l \oplus Ak$. Clearly, pixel $p_{u,v}$ belongs to subimage $\xi(l)$ for $p_{u,v}$ is in $\mathbf{P}_A^l$.

Let $i = \Gamma(k)$, and $j = \Omega(k)$. Then, $\mathbf{PE}_k$ is located at the ith row and the jth column of the PE array. In other words, it is PE (i,j). What remains to be shown now is that $p_{u,v}$ is pixel (i,j) of the subimage.

In fact, pixel $p_{u,v}$ is located at row $\Omega(u)$ and column $\Omega(v)$ of the subimage. In other words, $p_{u,v}$ is, in fact, pixel $(\Omega(u),\Omega(v))$ of the subimage.

Substituting the identity $A = [0^{\frac{n}{2}} 1^{\frac{n}{2}}]$ into the expressions for u and v yields

$$u = l_{\frac{n}{2}} \cdots l_{n-1} k_0 \cdots k_{\frac{n}{2}-1},$$

and

$$v = l_0 \cdots l_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}.$$

It is easy to see that

$$\Omega(l_{\frac{n}{2}} \cdots l_{n-1} k_0 \cdots k_{\frac{n}{2}-1}) = i,$$

and

$$\Omega(l_0 \cdots l_{\frac{n}{2}-1} k_{\frac{n}{2}} \cdots k_{n-1}) = j.$$

This completes the proof.

Q.E.D.

In viewing the PE's as a two-dimensional array, Corollary 3.5 claims that the architecture permits the PE's to process one subimage at a time with neighboring PE's being responsible for neighboring pixels of that subimage.

The next corollary claims that the elements of any mesh can be placed on the PE array in the desired order.

Corollary 3.6: For any given mesh, PE (i,j) can access pixel (i,j) of the mesh, for all $i,j \in \mathbf{I}_r$.

Proof: From Corollary 3.2, $\mathbf{P}_A^l$ corresponds to the lth mesh when A satisfies the condition $A = [1^{\frac{n}{2}}0^{\frac{n}{2}}]$. On the other hand, by Theorem 3.5, $\mathbf{PE}_k$ can read $p_{u,v}$ from $\mathbf{M}$, where $u = \xi(Al \oplus \bar{A}\xi(k))$, and $v = \bar{A}l \oplus A\xi(k)$. It is clear that pixel $p_{u,v}$ belongs to the lth mesh for pixel $p_{u,v}$ is in $\mathbf{P}_A^l$.

It can be shown that $\mathbf{PE}_k$ is PE (i,j), where $i = \Gamma(k)$ and $j = \Omega(k)$, respectively, and $p_{u,v}$ is pixel $(\Gamma(u),\Gamma(v))$ of the mesh-cover. It remains to show that $\Gamma(u) = i$ and $\Gamma(v) = j$.

Substituting the identity $A = [1^{\frac{n}{2}}0^{\frac{n}{2}}]$ into the expressions for u and v yields

$$u = k_0 \cdots k_{\frac{n}{2}-1} l_0 \cdots l_{\frac{n}{2}-1},$$

and

$$v = k_{\frac{n}{2}} \cdots k_{n-1} l_{\frac{n}{2}} \cdots l_{n-1}.$$

It is easy to see that

$$\Gamma(k_0 \cdots k_{\frac{n}{2}-1} l_0 \cdots l_{\frac{n}{2}-1}) = i,$$

and

$$\Gamma(k_{\frac{n}{2}} \cdots k_{n-1} l_{\frac{n}{2}} \cdots l_{n-1}) = j.$$

The corollary, therefore, follows.

Q.E.D.

In viewing the **PE's** as a two-dimensional array, Corollary 3.6 claims that the architecture allows each processing element to take care of a subimage with neighboring PE's being responsible for neighboring subimages in parallel.

The corollary below concerns the placement of any row of the image on PE's in the desired order.

Corollary 3.7: For any given row of the image, PE_i can access the kth pixel of that row for all $k \in I_N$.

Proof: From Corollary 3.3, P'_A corresponds to the $\xi(l)$th row of the image when A satisfies the condition $A = [1^n]$. From Theorem 3.5, PE_i can read $p_{u,v}$, where $u = \xi(Al \oplus \bar{A}k)$, and $v = \bar{A}l \oplus Ak$. Substituting A with $[1^n]$ into u and v will yield $u = \xi(l)$, and $v = k$. That is, PE_i can read $p_{\xi(l),k}$, which is the kth pixel of the $\xi(l)$th row of the image.

The corollary therefore follows.

Q.E.D.

In viewing the PE's as an one-dimensional array, Corollary 3.7 claims that the architecture allows the PE's to process any row of the image at a time with adjacent PE's being responsible for the adjacent pixels on that row.

The next corollary concerns the placement of any column of the image on the PE's in the desired order.

Corollary 3.8: For any given column of the image, PE_i can access the kth pixel of the column, for all $k \in I_N$.

Proof: By Corollary 3.4, $\mathbf{P}_A^l$ corresponds to the lth column of the image when integer A satisfies the condition $A = [0^n]$. From Theorem 3.5, $\mathbf{PE}_k$ can read $p_{u,v}$, where $u = \xi(Al \oplus \overline{A}\xi(k))$, and $v = \overline{A}l \oplus A\xi(k)$. Substituting A with $[0^n]$ into u and v will yield $u = k$, and $v = l$. That is, $\mathbf{PE}_k$ can read $p_{k,l}$, which is the kth pixel of the lth column of the image. This completes the proof.

Q.E.D.

Similarly, taking the PE's as an one-dimensional array, Corollary 3.8 claims that the architecture permits the PE's to process any column of the image $\mathbf{P}$ with adjacent PE's being responsible for adjacent pixels in that column. As can be seen from Corollaries 3.7 and 3.8, the corresponding elements of any two given row and column of the image can be brought into the same processing element by exactly two read operations. Such an architectural feature can be directly applied to computing the product of two $N \times N$ matrices in parallel.

As pointed out earlier, there are N possible N-partitions of $\mathbf{P}$ into $\mathbf{P}_A^l$. The foregoing discussions have shown four of them. More will be introduced in subsequent chapters when needed for parallel implementation of image algorithms.

3.7. The machine instructions I

There is no attempt to give a complete set of machine instructions for the proposed architecture at this point. Only those which are fundamental to understanding how the multiple-processor can be programmed for parallel image processing will be fully described in this section. Instructions fall roughly into the following four categories:

1. control unit instructions,
2. processing element instructions,
3. data movement instructions,

4. Input/output instructions.

The first category covers instructions that will be executed directly in the CUN. These include instructions used in calculating the addresses of operands in the local memories of the processing elements, in counting for program loops, and in controlling the order of execution of a program. The processing element instructions, as their name implies, are executed directly by the PE's. These instructions are used by the PE to perform arithmetic and logic operations and to access its local memory. The third category covers instructions which are used to move data between the parallel memory system **M** and the PE array, instructions which are used to move data among PE's through the Θ network, and instructions which are used to move data among PE's through the nearest neighbor connection. The input/output instructions are used to load/unload the parallel memory system **M**.

Returning to the third category, there are four instructions: read, write, transfer, and shift.

The read instruction

$$\mathbf{R}_i \leftarrow \mathbf{M}:(A, l, \alpha)$$

where $A, l \in \mathbf{I}_N$ and $\alpha \in \{0,1\}$, moves data from **M** to PE array as follows.

The execution of the read instruction consists of the following steps. First, the CUN connects $\mathbf{M}_k$ and register $\mathbf{R}_i$ of $\mathbf{PE}_k$ to network input k and network output k, respectively. Second, the address generating mechanism uses integers A and l to compute the expression $l \oplus Ak$ as the address for the kth memory module of **M**, for all $k \in \mathbf{I}_N$ in parallel. Third, the CUN constructs the routing vector of the Θ network as follows. If α is 0, then $0l$ is used as the routing vector to set up the Θ network to realize the mapping of network input k to network output $k \oplus l$. Thus, upon completion of the read instruction, register $\mathbf{R}_i$ of processing element k is loaded with the data stored in location $\overline{A}l \oplus Ak$ of memory module $k \oplus l$ for all

$k \in I_N$ in parallel. If α is 1, however, then $1\xi(l)$ is used as the routing vector to set up the Θ network instead. In this case, the read instruction loads register R_i of the kth processing element with the data stored in location $\bar{A}l \oplus A\xi(k)$ of memory module $\xi(k) \oplus l$ for all $k \in I_N$ in parallel.

The instruction can be used to move the elements of P_A^l from $\mathbf{M}$ and place them on PE array in the order as specified by Theorem 3.5. In particular, the instruction $R_i \leftarrow \mathbf{M}:(A, l, 0)$ loads register R_i of the kth processing element with $p_{\xi(Al \oplus \bar{A}k), \bar{A}l \oplus Ak}$ for all $k \in I_N$. On the other hand, the instruction $R_i \leftarrow \mathbf{M}:(A, l, 1)$ loads register R_i of the kth processing element with $p_{\xi(Al \oplus \bar{A}\xi(k)), \bar{A}l \oplus A\xi(k)}$ for all $k \in I_N$ in parallel.

The write instruction

$$R_i \rightarrow \mathbf{M}:(A, l, \alpha)$$

where $A, l \in I_N$ and $\alpha \in \{0,1\}$, move data from PE array to $\mathbf{M}$ as follows.

The execution of the write instruction consists of the following steps. First, the CUN connects M_k and register R_i of PE_k to network output k and network input k, respectively. Second, the address generating mechanism uses integers A and l to compute the expression $l \oplus Ak$ as the address for the kth memory module, for all $k \in I_N$ in parallel. Third, the CUN constructs the routing vector of the Θ network as follows. If α is 0, then $0l$ is used as the routing vector to control the Θ network. Upon termination of the write instruction, the contents of register R_i of processing element k is stored in location $\bar{A}l \oplus Ak$ of memory module $k \oplus l$ for all $k \in I_N$ in parallel. If α is 1, however, then $1l$ is used as the routing vector to set up the Θ network instead to realize the mapping of network input k to network output $\xi(k) \oplus l$. Accordingly, the write instruction stores the contents of register R_i of the kth processing element into location $\bar{A}l \oplus A\xi(k)$ of memory module $\xi(k) \oplus l$, for all $k \in I_N$ in parallel. As can be seen, the function of the write instruction is based on Theorems 3.5 and 3.6. On the one hand, the instruction $R_i \rightarrow$

$\mathbf{M}:(A, l, 0)$ will store the contents of register $\mathbf{R}_i$ of the kth processing element in place of pixel $p_{\xi(Al \oplus \overline{A}k), \overline{A}l \oplus Ak}$ for all $k \in \mathbf{I}_N$. On the other hand, the instruction $\mathbf{R}_i \rightarrow \mathbf{M}:(A, l, 1)$ will store the contents of register $\mathbf{R}_i$ of the kth processing element in place of pixel $p_{\xi(Al \oplus \overline{A}\xi(k)), \overline{A}l \oplus A\xi(k)}$ for all $k \in \mathbf{I}_N$ in parallel.

The transfer instruction

$$\mathbf{R}_i \leftarrow \mathbf{R}_j :(theta, l, \alpha)$$

where $l \in \mathbf{I}_N$ and $\alpha \in \{0,1\}$, shuffles data among the PE's through the Θ network as follows.

The execution of the transfer instruction consists of the following steps. First, the CUN connects registers $\mathbf{R}_j$ and $\mathbf{R}_i$ of the kth processing element to the network input terminal k and network output terminal k, respectively. Second, the CUN uses routing vector αl to set up the theta network. On completion of the transfer instruction, the contents of register $\mathbf{R}_j$ of a PE will be transferred to register $\mathbf{R}_i$ of an appropriate PE, according to the routing vector being used. In particular, the instruction $\mathbf{R}_i \leftarrow \mathbf{R}_j :(theta, l, 0)$ transfers the data stored in register $\mathbf{R}_j$ of $\mathbf{PE}_{k \oplus l}$ to register $\mathbf{R}_i$ of $\mathbf{PE}_k$, for all $k \in \mathbf{I}_N$ in parallel. On the other hand, the instruction $\mathbf{R}_i \leftarrow \mathbf{R}_j :(theta, l, 1)$ transfers the data stored in register $\mathbf{R}_j$ of $\mathbf{PE}_{\xi(k \oplus l)}$ to register $\mathbf{R}_i$ of $\mathbf{PE}_k$, for all $k \in \mathbf{I}_N$ in parallel.

The shift instruction

$$\mathbf{R}_i \leftarrow \mathbf{R}_j :(mesh, s)$$

where s is an integer and $0 \leq s \leq 7$, shifts data among the PE's through the nearest neighbor connection. The execution of the instruction involves the following steps. First, the CUN enables a nearest neighbor connection according to the function F_s defined previously. Second, the kth PE loads its $\mathbf{R}_i$ register with the contents of $\mathbf{R}_j$ of the $(F_s(k))$th PE, which is the sth neighboring PE, for all $k \in \mathbf{I}_N$ in parallel. If such a neighboring PE does not exist, then $\mathbf{R}_i$ is loaded with

zero.

In the fourth category, there are two instructions: input and output. The purpose of the input instruction is to provide a means by which the image **P** can be loaded into the parallel memory system **M** according to the storage scheme **S**. In principle, an input instruction can be designed to load $\mathbf{P}_A^I$ of image **P** into **M** such that **S** is satisfied, where integer A is a design factor. Thus, there could be N possible different input instructions to load $\mathbf{P}_A^I$, one for each fixed integer $A \in \mathbf{I}_N$. Once the value of integer A is chosen, the function of the corresponding input instruction is determined completely. Then, through a loop of N such input instructions the entire image **P** can be loaded into **M**, thereby storing **P** in **M** under the storage scheme **S**. Recall that when A satisfies the condition $A = [1^n]$ the resulting $\mathbf{P}_A^I$ corresponds to the $\xi(I)$th row of the image. Consequently, the following input instruction is defined to load one row of the image into **M**.

The input instruction

IN(i)

where $i \in \mathbf{I}_N$ loads the ith row of image **P** into the parallel memory system as follows. The execution of the instruction consists of two steps. The CUN first moves the ith row of the image from the host computer onto the PE's such that $\mathbf{R}_0$ of $\mathbf{PE}_j$ receives the jth pixel of the row, i.e., $p_{i,j}$. Then, the write instruction $\mathbf{R}_0 \rightarrow \mathbf{M}:([1^n], \xi(i), 0)$ is executed to store pixel $p_{i,j}$ in $\mathbf{M}_{\xi(i) \oplus j, j}$ for all $j \in \mathbf{I}_N$.

The output instruction

OUT(i)

where $i \in \mathbf{I}_N$ outputs the ith row of the image in **M** as follows. The execution of the instruction consists of also two steps. First, the read instruction $\mathbf{R}_0 \leftarrow \mathbf{M}:([1^n], \xi(i), 0)$ is executed to read the jth pixel of the ith row into $\mathbf{R}_0$ of $\mathbf{PE}_j$ for all $j \in \mathbf{I}_N$. The CUN then moves the row out of the PE's to the host computer.

In the design of input/output instructions, there exists a spectrum of design alternatives. For example, it might be useful to input and output images on the subimage basis as well. When this is the case, the desired input/output instructions can be added.

Being of the SIMD type, the proposed multiple-processor architecture cannot support two independent instruction streams simultaneously. Within the constraints imposed, the two cases must be done sequentially. A means must be provided to disable a selected subset of PE's, and execute an execution sequence in parallel on the subset that remains active. This way a stream of instructions can be executed on the PE's that are active. Then another stream of instructions is executed on the complementary set of PE's. The two separate instruction streams merge into a single stream after their execution.

To implement this type of masking facility, each processing element has a n-bit mask register to hold its index as shown in Fig. 3.4, and each instruction broadcast by the CUN contains a mask field of n bits. When the contents of the mask register in PE_i match the mask field, PE_i obeys the instruction. When there is a missmatch, PE_i does nothing. For example, the processing element instruction "$R_2 = R_0 + R_1 \quad (\phi^{n-1}0)$" will be executed by only the even number indexed processing elements. The symbol ϕ in the mask field of the instruction stands for the *don't care* sign. Likewise, the instruction "$R_2 = R_0 - R_1 \quad (\phi^{n-1}1)$" will be executed by only the odd number indexed processing elements. Instructions whose mask field consists of entirely don't care signs are obeyed by all PE's. It is apparent that the masking facility will also allow the CUN to read a data item from a particular PE into its own register by enabling exactly one PE to be active.

3.8. Conclusion

A multiple-processor architecture for image processing has been presented in this chapter. The architecture implements a set of image partitioning strategies by allowing 1) various N-partitions of image $\mathbf{P}$; 2) parallel access of any of parts of an N-partition of $\mathbf{P}$ without memory conflict; and 3) placement of the elements of a part on the PE array in useful orders. The value of such an architectural feature lies in its use for the dynamic supporting of structural matching for a variety of image processing algorithms. The next two chapters show how this computer architecture can be used effectively to support image processing tasks within the three classes defined in Chapter 2.

CHAPTER 4

ALGORITHMS FOR NEIGHBORHOOD OPERATIONS

Based on the computer architecture proposed in Chapter 3, this chapter considers efficient parallel implementations of neighborhood operations. It is intended to demonstrate the dynamic algorithm-architecture matching capability of the architecture in achieving the computational efficiency for both local and global neighborhood operations.

4.1. Introduction

As mentioned in Chapter 2, local neighborhood operations and global neighborhood operations are diverse in their computational structures, and the parallelism in an image processing task can be exploited by an image processor if the computing structure of the machine matches the computational structure of the task. It will be shown that by choosing appropriate image partitioning strategies, the proposed architecture is able to support dynamically the structural matching according to the type of neighborhood operations being performed. To demonstrate such an architectural feature, the use of the architecture for performing convolution will be considered. Although convolution (linear filters) is not the only operation in the family of neighborhood operations, which include also a variety of nonlinear filters such as median, direction sensitive filters, etc., the communication structure of computing convolution is common to all neighborhood operations in nature.

4.2. A parallel algorithm for 3×3 convolution

This section will present a parallel algorithm for the convolution of image **P** and a 3×3 kernel **W**:

$$g_{i,j} = \frac{1}{9} \sum_{h=-1}^{1} \sum_{v=-1}^{1} w_{h,v} \times p_{i-h,j-v}$$

where

$$\mathbf{W} = \begin{pmatrix} w_{-1,1} & w_{0,1} & w_{1,1} \\ w_{-1,0} & w_{0,0} & w_{1,0} \\ w_{-1,-1} & w_{0,-1} & w_{1,-1} \end{pmatrix}.$$

As was stated earlier, the image **P** can be conceptually decomposed into an array of $r \times r$ subimages with each subimage having eight immediate neighbors except those at an edge of the array. The following notation is used in presenting the parallel convolution algorithm:

$\mathbf{S}_{i,j}$ Subimage (i,j) of image **P**, where $i,j \in \mathbf{I}_r$.

$\mathbf{S}_{i,j}^{u,v}$ Pixel (u,v) of $\mathbf{S}_{i,j}$, where $u,v \in \mathbf{I}_r$.

$(\mathbf{S}_{i,j})_k$ The kth neighboring subimage of $\mathbf{S}_{i,j}$, where $0 \leq k \leq 7$. The ordering is defined as follows:

$$\begin{matrix} (\mathbf{S}_{i,j})_3 & (\mathbf{S}_{i,j})_4 & (\mathbf{S}_{i,j})_5 \\ (\mathbf{S}_{i,j})_2 & \mathbf{S}_{i,j} & (\mathbf{S}_{i,j})_6 \\ (\mathbf{S}_{i,j})_1 & (\mathbf{S}_{i,j})_0 & (\mathbf{S}_{i,j})_7 \end{matrix}.$$

$(\mathbf{S}_{i,j}^{u,v})_k$ The kth neighboring pixel of $\mathbf{S}_{i,j}^{u,v}$ with the same ordering as defined above.

$\mathbf{PE}_{i,j}$ PE (i,j) of the PE array, where $i,j \in \mathbf{I}_r$.

$(PE_{i,j})_k$ The kth neighbor of $PE_{i,j}$ with the same ordering as above.

By Corollary 3.5, the architecture permits the placement of any subimage of **P** on the PE array, by one read operation, in such a way that neighboring pixels of the subimage are placed in neighboring processing elements of the PE array. Having the subimage placed on the PE array, the nearest neighbor connection of the PE's allows all processing elements to directly access pixel values in the 3×3 window from neighboring processing elements in parallel, except those located at an edge of the PE array, which are called edge processing elements. Each edge processing element with a border pixel of the subimage also needs values of some pixels which are not in the subimage in order to generate the corresponding border pixel for the new image **G**. A parallel local convolution algorithm could have been implemented on SIMD machines in a straightforward manner if the edge problem did not exit. The problem is essentially imposed by the constraint that the number of processors is smaller than the number of pixels in the image to be processed in the real world. The difficulty associated with the edge problem is that the values of certain pixels of the image cannot be obtained by edge processing elements in synchronism with nonedge processing elements during the computation, thereby losing parallelism. The parallel convolution algorithm to follow solves the edge problem as follows.

According to Corollary 3.6, any mesh of **P** can be placed on the PE array, by one read operation, in such a way that pixel (i,j) of the mesh is placed in PE (i,j) for all $i,j \in I_r$. Recall that pixel (i,j) of the lth mesh is pixel $(\Gamma(l),\Omega(l))$ of subimage (i,j) of **P**. In other words, $S_{i,j}^{\Gamma(l),\Omega(l)}$ is placed in $PE_{i,j}$. Thus, choosing the value of l appropriately, the architecture allows $PE_{i,j}$ to obtain the value of each desired pixel outside $S_{i,j}$ by one read operation followed by a shift operation in synchronism with other processing elements. Since every processing element is

doing the same thing, there is no loss of parallelism. The parallel convolution algorithm consists of the following three phases.

In the first phase of the algorithm, each processing element carries out a weighted-sum computation with respect to a border pixel of a subimage by taking into account only those neighbors of the border pixel which are not in the subimage, and stores the result in its local memory. This process is repeated $4(r-1)$ times for the $4(r-1)$ border pixels. The data thus obtained will be used during the third phase of the algorithm by the processing element to generate the final output **G**.

The process of the first phase is illustrated pictorially in Fig. 4.1 by a matrix which depicts a "snapshot" of the N processing elements accessing the first border pixel of their responsible subimages for the case of $N = 16$.

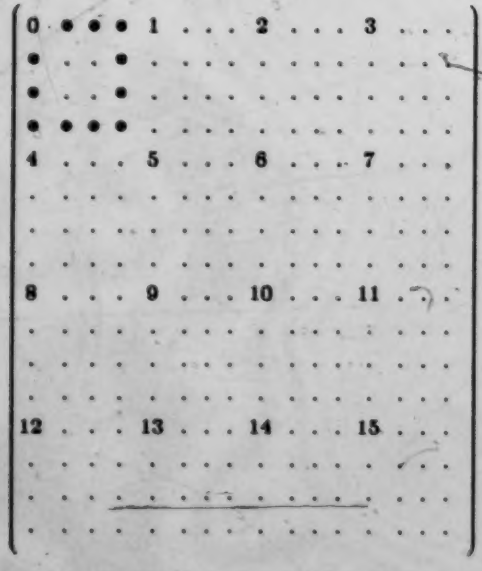

Fig. 4.1. $PE_{i,j}$ accessing $S_{i,j}^{0,0}$.

The integer at the entry located at the ith row and the jth column of the matrix referred to as the access matrix represents the number of the processing

element by which pixel $p_{i,j}$ is being accessed, and a dot in an entry simply means that the corresponding pixel is not accessed by any processing element at the present. For example, PE_0 is accessing the border pixel $S_{0,0}^{0,0}$ in synchronism with other PE's. As PE_0 reads in this border pixel, computes the partial result for it, and stores the partial result in its local memory, every other processing element is performing the same operation on its responsible border pixel in parallel. The process will be repeated twelve times for the twelve border pixels, where a border pixel of $S_{0,0}$ is denoted by • in the access matrix.

To be more precise, let $v \in I_r$, and let each processing element make use of $4 \times r$ locations of its local memory which are represented as four arrays: B_0, B_1, B_2, and B_3, respectively. In particular, array B_0 is used for the computations with respect to border pixels at row $r-1$ of the subimage with $B_{0,v}$ for pixel $(r-1,v)$ of the subimage. Array B_1 is used for the computations with respect to border pixels at column 0 of the subimage with $B_{1,v}$ for pixel $(v,0)$ of the subimage. Array B_2 is used for the computations with respect to border pixels at row 0 of the subimage with $B_{2,r-v-1}$ for pixel $(0,v)$ of the subimage. Array B_3 is used for the computations with respect to border pixels at column $r-1$ of the subimage with $B_{3,r-v-1}$ for pixel $(v,r-1)$ of the subimage.

Next, express the convolution kernel W as follows

$$W = \begin{pmatrix} \omega_3 & \omega_4 & \omega_5 \\ \omega_2 & \omega & \omega_6 \\ \omega_1 & \omega_0 & \omega_7 \end{pmatrix}$$

for purposes of description of the algorithm only. As the output of the first phase of the algorithm, the contents of the four arrays are as follows:

$$\mathbf{B}_{0,v} = \begin{cases} \sum_{k=0,1,7} (\mathbf{S}_{i,j}^{r-1,v})_k \times \omega_k & \text{if } 0 < v < r-1 \\ \sum_{k=0,1,5,6,7} (\mathbf{S}_{i,j}^{r-1,r-1})_k \times \omega_k & \text{if } v = r-1 \end{cases}$$

$$\mathbf{B}_{1,v} = \begin{cases} \sum_{k=1,2,3} (\mathbf{S}_{i,j}^{r,v})_k \times \omega_k & \text{if } 0 < v < r-1 \\ \sum_{k=0,1,2,3,7} (\mathbf{S}_{i,j}^{r-1,v})_k \times \omega_k & \text{if } v = r-1 \end{cases}$$

$$\mathbf{B}_{2,v} = \begin{cases} \sum_{k=3,4,5} (\mathbf{S}_{i,j}^{v,r-v-1})_k \times \omega_k & \text{if } 0 < v < r-1 \\ \sum_{k=1,2,3,4,5} (\mathbf{S}_{i,j}^{v,v})_k \times \omega_k & \text{if } v = r-1 \end{cases}$$

$$\mathbf{B}_{3,v} = \begin{cases} \sum_{k=5,6,7} (\mathbf{S}_{i,j}^{r-v-1,r-1})_k \times \omega_k & \text{if } 0 < v < r-1 \\ \sum_{k=3,4,5,6,7} (\mathbf{S}_{i,j}^{v,r-1})_k \times \omega_k & \text{if } v = r-1 \end{cases}$$

The first phase of the algorithm is now presented formally below as algorithm **Border**:

Algorithm Border

```
    begin
                 n/2 n/2
1.     A := 1    0   ;
2.     for u := 0 to 3 do
3.       for v := 1 to r −1 do
           begin
4.           case u of
               0: l := v;
               1: l := r×v + r − 1;
               2: l := r(r − 1) + r − v − 1;
               3: l := r(r −v − 1);
             end;
5.           R0 ← M :(A, l, 1);
             if v = r − 1 then
               begin
6.               R1 ← R0 :(mesh, 2u);
7.               for k := −1 to 0 do
                     B(u,v+k) += R1×ω(2u+k);
8.               R1 ← R0 :(mesh, 2u + 1);
9.               B(u,0) += R1×ω(2u+1);
10.              R1 ← R0 :(mesh, 2u + 2);
11.              for k := 0 to 1 do
                     B(u+1,k) += R1×ω(2u+2+k);
               end else
               begin
12.              R1 ← R0 :(mesh, 2u);
13.              for k := −1 to 1 do
                     B(u,v+k) += R1×ω(2u+k);
               end;
           end;
14.    for u := 0 to 3 do
           B(u+1,r−1) += B(u,0);
    end;
```

The action of algorithm **Border** can be described briefly as follows. Line 1 initializes A according to Corollary 3.6. It is important to notice that the read instruction at line 5, which moves the lth mesh of $\mathbf{P}$ from $\mathbf{M}$ onto the PE array, will load register $\mathbf{R}_0$ of $\mathbf{PE}_{i,j}$ with $\mathbf{S}_{i,j}^{\Gamma(l),\Omega(l)}$ for all $i,j \in \mathbf{I}$, in parallel.

Consider the first iteration of the for loop at line 2 ($u = 0$). During the first iteration of the for loop at line 3 ($v = 1$), after line 5 is executed, register $\mathbf{R}_0$ of $\mathbf{PE}_{i,j}$ contains $\mathbf{S}_{i,j}^{0,1}$, which is a neighbor of three border pixels in $(\mathbf{S}_{i,j})_4$. It follows that $(\mathbf{PE}_{i,j})_0$ has its $\mathbf{R}_0$ containing a pixel of $(\mathbf{S}_{i,j})_0$, which is neighbor 7 of $\mathbf{S}_{i,j}^{-1,0}$,

neighbor 0 of $S^{[-1,1]}_{i,j}$, and neighbor 1 of $S^{[-1,2]}_{i,j}$, respectively. The shift instruction at line 12 then moves this pixel into R_1 of $PE_{i,j}$ from R_0 of $(PE_{i,j})_0$, and the for loop at line 13 augments $B_{0,0}$, $B_{0,1}$ and $B_{0,2}$ by the amounts of $(S^{[-1,0]}_{i,j})_7 \times \omega_7$, $(S^{[-1,1]}_{i,j})_0 \times \omega_0$, and $(S^{[-1,2]}_{i,j})_1 \times \omega_1$, respectively. Therefore, the weighted contributions of the pixel to its three neighbors in $S_{i,j}$ are recorded by $PE_{i,j}$ in its local memory.

In the succeeding iterations of the for loop at line 3, u specifies $S^{[-1,u]}_{i,j}$ to be read, and memory locations $B_{0,u-1}$, $B_{0,u}$, and $B_{0,u+1}$ to be augmented by $PE_{i,j}$. The whole process is then repeated, and the loop executed $r-2$ times. At the last iteration of the for loop ($v = r-1$), after line 5 is executed, register R_0 of $PE_{i,j}$ contains $S^{0,r-1}_{i,j}$, which is a neighbor of five pixels outside $S_{i,j}$. Firstly, R_0 of $(PE_{i,j})_0$ contains a pixel of $(S_{i,j})_0$, which is neighbor 7 of $S^{[-1,r-2]}_{i,j}$, and neighbor 0 of $S^{[-1,r-1]}_{i,j}$. This pixel is then moved into R_1 of $PE_{i,j}$ at line 6, and the loop at line 7 adds in effect $(S^{[-1,r-2]}_{i,j})_7 \times \omega_7$, and $(S^{[-1,r-1]}_{i,j})_0 \times \omega_0$ to $B_{0,r-2}$ and $B_{0,r-1}$, correspondingly. Secondly, R_0 of $(PE_{i,j})_1$ contains a pixel of $(S_{i,j})_1$ which is neighbor 1 of $S^{[-1,0]}_{i,j}$. This pixel is moved into R_1 of $PE_{i,j}$ at line 8. Then, line 9 adds $(S^{[-1,0]}_{i,j})_1 \times \omega_1$ to $B_{0,0}$. Finally, R_0 of $(PE_{i,j})_2$ contains a pixel of $(S_{i,j})_2$ which is neighbor 2 of $S^{0,0}_{i,j}$, and neighbor 3 of $S^{1,0}_{i,j}$, respectively. This pixel is then moved into R_1 of $PE_{i,j}$ at line 10, and the loop at line 11 adds $(S^{0,0}_{i,j})_2 \times \omega_2$, and $(S^{1,0}_{i,j})_3 \times \omega_3$ to $B_{1,0}$ and $B_{1,1}$, respectively. Thus, the weighted contributions of the three pixels outside $S_{i,j}$ are stored by $PE_{i,j}$.

In succeeding iterations of the for loop at line 2, u specifies which border to process with $u = 0, 1, 2, 3$ for bottom, left, right, and top borders, respectively. The whole process is then repeated, and the loop is executed 4 times.

It can be shown that at the termination of algorithm **Border** $PE_{i,j}$ generates the contents of the four arrays as defined for all $i,j \in I$, in parallel. Notice that image **P** in **M** remains unchanged up to this point, for no write operation has ever been used during the first phase. The second phase of the algorithm is, however, to change **P** to a new image from which image **G** can be computed by

the third phase.

In the second phase, a subimage of **P** is placed on the PE array first. $PE_{i,j}$ provided with pixel (i,j) of the subimage accesses neighbors of the pixel in the 3×3 window from its neighboring processing elements through the nearest neighbor connection, where edge processing element is supplied with merely a zero by those neighbors that are not in existence, and carries out the weighted-sum computation with respect to the pixel provided. This way, each nonedge processing element, which is provided with an interior pixel of the subimage, computes a new pixel from the interior pixel and its eight neighbors. In contrast, each edge processing element, which is provided with a border pixel of the subimage, computes a new pixel from the border pixel and its neighbors in the subimage. The subimage is then replaced by the newly computed subimage. The process is repeated N times for the N subimages of **P**.

The process of the second phase is illustrated pictorially in Fig. 4.2 by the access matrix which depicts a "snapshot" of the N PE's working on the last subimage of **P** for the case of $N = 16$. As shown, PE_5, which is a nonedge processing element, is working on $S_{3,3}^{1,1}$ to compute the new value for the pixel, while PE_{15}, which is an edge processing element, is working on $S_{3,3}^{3,3}$ to compute the partial value for the pixel. Depicted by the access matrix above is another fact that each interior pixel of the first 15 subimages has its new value completely computed at this point in time, where an interior pixel is denoted by • in the access matrix.

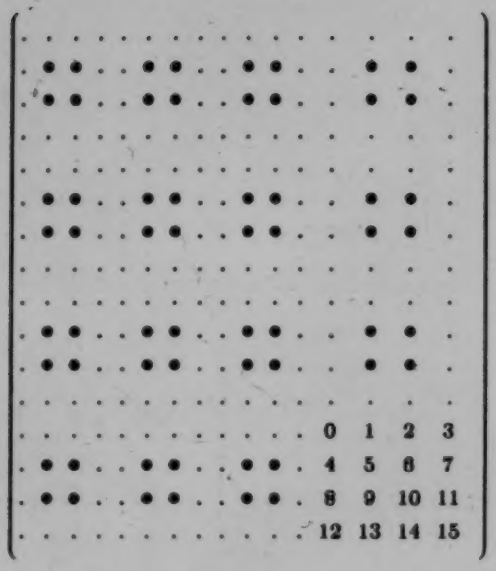

Fig. 4.2. PE's processing $S_{3,3}$.

The second phase of the algorithm is now formally represented below as algorithm **Interior**.

Algorithm Interior

```
    begin
1.      A := 0^(n/2) 1^(n/2);
2.      for l := 0 to N − 1 do
          begin
3.          R0 ← M :(A, l, 0);
4.          R2 := R0×ω;
5.          for k := 0 to 7 do
              begin
6.              R1 ← R0 :(mesh, k);
7.              R2 += R1×ωk;
              end;
8.          R2 → M :(A, l, 0);
          end;
    end;
```

The action of algorithm **Interior** can be briefly described as follows. Line 1

initializes A according to Corollary 3.5. It is important to observe that the read instruction, which places the $\xi(l)$th subimage of $\mathbf{P}$ on PE array, reads $\mathbf{S}^{i,j}_{\xi(l),\eta(l)}$ into $\mathbf{R}_0$ of $\mathbf{PE}_{i,j}$ for all $i,j \in \mathbf{I}_r$. During the first iteration of the loop at line 2 ($l = 0$), after line 3 is executed, $\mathbf{R}_0$ of $\mathbf{PE}_{i,j}$ contains $\mathbf{S}^{i,j}_{0,0}$. Therefore, line 4 loads $\mathbf{R}_2$ with $\mathbf{S}^{i,j}_{0,0} \times \omega$. Notice that $\mathbf{R}_0$ of $(\mathbf{PE}_{i,j})_k$ contains $(\mathbf{S}^{i,j}_{0,0})_k$ for $0 \leq k \leq 7$. The $\mathbf{PE}_{i,j}$ then obtains $(\mathbf{S}^{i,j}_{0,0})_k$ from its kth neighboring processing element by the shift instruction at line 6, and adds $(\mathbf{S}^{i,j}_{0,0})_k \times \omega_k$ to the $\mathbf{R}_2$ register at line 7 at the kth iteration of the loop at line 5. In effect, each nonedge processing element has taken into account all the eight neighbors of the interior pixel in the computation while each edge processing element has considered only five neighbors of the border pixel in the subimage. The write instruction at line 8 replaces $\mathbf{S}^{i,j}_{0,0}$ by the contents of $\mathbf{R}_2$ of $\mathbf{PE}_{i,j}$ for all $i,j \in \mathbf{I}_r$ simultaneously. In succeeding iterations of the loop, l specifies the next subimage to be processed. The whole process is then repeated, and the loop executed N times.

On termination of phase 2, the newly computed image in $\mathbf{M}$ is not the same as the desired image $\mathbf{G}$ for the values of the border pixels in every subimage of the new image need be further augmented. Phase 3 of the algorithm, therefore, rectifies values of these border pixels accordingly.

In the third phase, $\mathbf{PE}_{i,j}$ augments the value of a border pixel of $\mathbf{S}_{i,j}$ of the new image computed by phase 2 by the amount which has been computed and recorded by phase 1 with respect to that border pixel, for all $i,j \in \mathbf{I}_r$. The process is repeated $4(r-1)$ times for all border pixels of every subimage of the new image. After phase 3 is finished, the image in $\mathbf{M}$ is precisely the desired image $\mathbf{G}$.

The process of the third phase is illustrated in Fig. 4.3 by the access matrix which depicts a "snapshot" of the N PE's rectifying the last border pixel of their subimages for the case of $N = 16$. As demonstrated by the access matrix, $\mathbf{PE}_0$ is augumenting the edge pixel $\mathbf{S}^{3,3}_{0,0}$ in parallel with other processing elements.

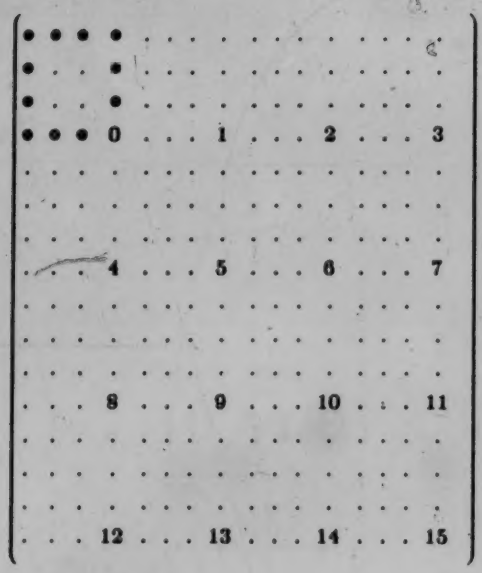

Fig. 4.3. $PE_{i,j}$ rectifying $S^{3,3}_{i,j}$.

Phase 3 is now presented below as Algorithm **Rectify**.

Algorithm Rectify

```
    begin
1.    A := 1^(r/2) 0^(r/2);
2.    for u := 0 to 3 do
3.      for v := 1 to r − 1 do
          begin
4.          case u of
              0: l := v;
              1: l := r×v + r − 1;
              2: l := r(r − 1) + r − v − 1;
              3: l := r(r − v − 1);
            end;
5.          R0 ← M :(A, l, 1);
6.          k := u + 2 mod 4;
7.          R2 := R0 + B(k, r−v−1);
8.          R2 → M :(A, l, 1);
          end;
    end;
```

The action of algorithm **Rectify** can be described as follows. Line 1 initializes A according to Corollary 3.6. The read instruction reads $S_{i,j}^{\lceil (r-1)/2 \rceil, \lceil (r-1)/2 \rceil}$ of the new image into R_0 of $PE_{i,j}$ for all $i,j \in I_n$. Consider the first iteration of the loop at line 2, i.e. $u = 0$. At the first iteration of the loop at line 3 ($v = 1$), after line 4 is executed, R_0 of $PE_{i,j}$ contains $S_{i,j}^{0,1}$ to be rectified. On the other hand, $B_{2,r-2}$ has been used by the processing element in phase 1 to store $\sum_{k=2}^{r}(S_{i,j}^{0,1})_k \times \omega_k$ which is precisely the amount to be added to $S_{i,j}^{0,1}$ for rectification. Line 7 stores the sum of $S_{i,j}^{0,1}$ and this amount in register R_2. Therefore, the write instruction at line 8 replaces $S_{i,j}^{0,1}$ by the sum computed, thereby, rectifying one border pixel of $S_{i,j}$ of the new image. In succeeding iterations of the loop at line 3, v specifies $S_{i,j}^{0,v}$ to be rectified, and the corresponding memory $B_{2,r-v-1}$ to be used. The whole process is then repeated, and the loop executed $r-1$ times to rectify $r-1$ border pixels at the top border of $S_{i,j}$ of the new image. In succeeding iterations of the loop at line 2, u specifies the next border to rectify with $u = 0, 1, 2, 3$ for top, right, bottom, and left borders, respectively. The process is then repeated, and the loop executed 4 time.

Now, the parallel convolution algorithm is represented below as Algorithm A.

Algorithm A

```
    begin
1.    Border;
2.    Interior;
3.    Rectify;
    end;
```

Theorem 4.1: Algorithm A computes the convolution of two images P and W in place.

Proof: The correctness of algorithm A will be proven by showing that at the completion of the algorithm, each $p_{x,y}$ is replaced by $g_{x,y} = p_{x,y} \times \omega + \sum_{k=0}^{7} (p_{x,y})_k \times \omega_k$, where $x,y \in \mathbf{I}_N$, and $(p_{x,y})_k$ is the kth neighbor of pixel $p_{x,y}$. Denote by σ pixel $g_{x,y}$. It follows that σ belongs to subimage $(\Gamma(x),\Gamma(y))$ of **G**, and is pixel $(\Omega(x),\Omega(y))$ of the subimage. Obviously, σ can be either an interior pixel or a border pixel of the subimage. Now, consider these two cases separately.

Case 1: σ is an interior pixel.

Clearly, neither algorithm **Border** nor algorithm **Rectify** compute σ. Algorithm **Interior**, however, does so when $p_{x,y}$ is placed in $\mathbf{PE}_{i,j}$, where $i = \Omega(x)$ and $j = \Omega(y)$. This processing element computes σ from the eight neighbors of $p_{x,y}$ which are in the eight neighbors of the processing elements then.

Case 2: σ is a border pixel of the subimage.

In fact, σ is a product of the cooperative efforts of **Border**, **Interior**, and **Rectify**. Firstly, algorithm **Border** uses $\mathbf{PE}_{i,j}$ to compute the first part of σ by taking into account only the neighbors of $p_{x,y}$ that are not in the subimage, where $i = \Gamma(x)$ and $j = \Gamma(y)$. Secondly, algorithm **Interior** uses $\mathbf{PE}_{i,j}$, which is provided with $p_{x,y}$ then, to compute the second part of σ by considering only the neighbors of $p_{x,y}$ which are in the subimage, where $i = \Omega(x)$ and $j = \Omega(y)$. Finally, algorithm **Rectify** uses $\mathbf{PE}_{i,j}$ to add the two parts together to form σ, where $i = \Gamma(x)$ and $j = \Gamma(y)$.

Q.E.D.

Measuring in terms of the number of read/write instructions, shift instructions, and processing element arithmetic instructions, the following theorem gives the time complexity of algorithm A.

Theorem 4.2: Algorithm A is of time complexity $O(N)$.

Proof: Algorithm **Border** takes $4(r-1)$ read operations, $4(r+1)$ shift operations, and $12(r-1)$ arithmetic operations. Algorithm **Interior** takes N read operations, $8N$ shift operations, $17N$ arithmetic operations, and N write operations. Algorithm **Rectify** takes $4(r-1)$ read operations, $4(r-1)$ arithmetic operations, and $4(r-1)$ write operations. In total, the cost of the algorithm is $2N + 12(r-1)$ read/write operations, $8N + 4(r+1)$ shift operations, and $17N + 16(r-1)$ arithmetic operations.

Q.E.D.

4.3. A parallel algorithm for global convolution

As has been demonstrated by Algorithm **Interior** in the last section, partitioning $\mathbf{P}$ into subimages according to Corollary 3.5 and using the PE array to process these subimages one at a time in parallel can exploit a high degree of parallelism for the convolution with 3×3 kernel in computing interior pixels. Accessing larger neighborhood requires shifting of data which seems to increase the number of shift instructions with at least a factor of two compared to 3×3 neighborhood access. Apart from the edge problem, it does not allow efficient access from the neighborhood corresponding to a sparse convolution kernel. The technique used in algorithm **Border** to access from 3×3 neighborhoods that overlay adjacent subimage can be extended as follows to allow efficient access from larger neighborhoods corresponding even to sparse kernels.

According to Corollary 3.6, for any $l \in \mathbf{I}_N$, the lth mesh of $\mathbf{P}$ can be placed on PE array such that $\mathbf{PE}_{i,j}$ is provided with pixel (u,v) of $\mathbf{S}_{i,j}$ for all $i,j \in \mathbf{I}_r$, where $u = \Gamma(l)$ and $v = \Omega(l)$. Through the nearest neighbor connection, when $\mathbf{PE}_{i,j}$ accesses a pixel from a neighborhood that overlays adjacent subimage, say, pixel (u,v) of $(\mathbf{S}_{i,j})_3$ which is in $(\mathbf{PE}_{i,j})_3$, $(\mathbf{PE}_{i,j})_2$ is doing the same accessing pixel (u,v) of

$S_{i,j}$ in $PE_{i,j}$. It is important to observe that as long as the size of the neighborhood does not exceed $2r$, any pixel in the neighborhood can be obtained by a read instruction followed by a shift instruction. When the kernel is a sparse one, each processing element needs only to access those that correspond to the nonzero entries of the kernel from its neighbors.

Therefore, a parallel convolution with kernel larger than 3×3 can be done in a straightforward way on the proposed architecture. A very general parallel algorithm for this class of operations is outlined below as algorithm **B** consisting of three steps.

Algorithm B

begin

1. For all $i,j \in I_r$, $PE_{i,j}$ loads its local memory with $S_{i,j}$ of image **P** in parallel.
2. For all $i,j \in I_r$, $PE_{i,j}$ convolves $S_{i,j}$ with the kernel, where neighborhoods that overlay adjacent subimages are accessed through the nearest neighbor connection in parallel.
3. For all $i,j \in I_r$, $PE_{i,j}$ stores the new subimage computed in step 2 in place of $S_{i,j}$ of **P** in **M** in parallel.

end.

It should be clear that the first step of the algorithm takes N read instructions to move the N subimages of **P** into the N processing elements, and the third step of the algorithm requires N write instructions to store the new image back into **M** in place of image **P**. At the second step of the algorithm, each processing element carries out a sequential algorithm for the convolution of its responsible subimage and the kernel. The sequential algorithm can be the same as the best known uniprocessor convolution algorithm except that it needs additional shift instructions to obtain pixels which are not in the subimage.

In fact, algorithm **B** essentially resembles the well known parallel convolution algorithm using a system of N processors with each processor having a local memory for its subimage, and connected directly to its immediate eight neighbors. Such a system is widely believed to be the most suitable computing structure for global neighborhood operations in the literature. The important point to make here is that the proposed architecture is versatile enough to support global neighborhood operations as efficiently.

It should be pointed out that the so-called pyramid processing can also be done on the proposed architecture with a high degree of parallelism. The mapping of the pyramid data structure to the proposed architecture can be accomplished as follows. The parallel memory system **M** will be used to store a pyramid of $\frac{n}{2}$ layers, where the bottom layer (layer 1) is of size $\frac{N}{2}\times\frac{N}{2}$ image and the top layer (layer $\frac{n}{2}$) is of size $r\times r$ image.

Let each layer be partitioned into N subimages. It follows that layer k of the pyramid has N subimages of size $\frac{r}{2^k}\times\frac{r}{2^k}$ each. Now, index the subimages of each layer in the same way as the subimages of image **P** are indexed. Denote by $\mathbf{S}^k_{i,j}$ subimage (i,j) of layer k of the pyramid. By Theorems 3.5, 3.6, and 3.7, the pyramid can be stored in **M** in such a way that $\mathbf{PE}_{i,j}$ can process $\mathbf{S}^k_{i,j}$ for any $1\leq k\leq\frac{n}{2}$ and all $i,j\in\mathbf{I}$, in parallel.

Consequently, pyramid processing can be achieved by applying the N PE's to one layer of the pyramid at a time in parallel while each layer is processed in sequence in either bottom-up or top-down fashion. However, the top of the pyramid above layer $\frac{n}{2}$ has to be handled outside the PE array.

4.4. Conclusion

Based on the proposed architecture, two parallel algorithms for neighborhood operations have been presented in this chapter. The mapping of the pyramid data structure to the proposed architecture has been briefly outlined.

Algorithm **A** can exploit a high degree of parallelism for local neighborhood operations in general, although described is only a 3×3 convolution operation. The edge problem is cleanly solved without loosing parallelism by making use of two image partitioning strategies in combination. As a consequence, the nearest neighbor connection hardware is fully utilized for each pixel of **G** in the computation. The algorithm has been presented in detail to illustrate how the proposed multiple-processor can be programmed for implementing parallel algorithms. The read, write, as well as shift instructions defined in Chapter 3 play an important role in this regard, for a thorough understanding of them among others helps prove the correctness of the parallel algorithm being programmed.

On the other hand, algorithm **B** is presented in very general terms to demonstrate that the architecture can dynamically lend itself to the most suitable computing structure for global neighborhood operations. As has been stated, as far as the kernel size does not exceed $2r$, each processing element can obtain any pixel from neighborhoods that overlay adjacent subimages by a read instruction followed by a shift instruction.

Clearly, the mapping of the pyramid data structure to the proposed architecture allows efficient implementations of pyramid algorithms.

CHAPTER 5

ALGORITHMS FOR GLOBAL TRANSFORMS

This chapter will focus on the efficient implementation of parallel algorithms for two operations in the category of global transforms on the proposed architecture. It is intended to demonstrate the point that it is the dynamic structural matching capability of the architecture that provides algorithm efficiency.

5.1. Introduction

It has been recognized that image processing tasks in the category of global transforms possess an entirely different communication structure from that of neighborhood operations. For instance, the butterfly operation described in Chapter 2 applies, at the sth iteration, to two elements of T_{s-1} which are $\frac{N}{2^s}$ elements apart to generate T_s. Yet, the matrix multiplication has another computational structure, where each element of every row of one matrix has to be paired with the corresponding element of every column of another matrix.

Section 5.2 presents a parallel 2-D FFT algorithm to show that the proposed architecture is able to match the computational structure effectively. Then, a parallel matrix multiplication algorithm is described in Section 5.3 to illustrate that the architecture is also capable of matching the computational structure as well. As will be seen, the key to achieve the algorithm-architecture matching is the use of pertinent image partitioning strategies.

5.2. 2-D Fast Fourier Transform

Taking the standard approach to computing the 2-D DFT of image **P** using row/column decomposing technique as described in Chapter 2, a parallel algorithm for the 1-D FFT to be presented is the key in implementing a parallel algorithm for the 2-D FFT. In order to match the architecture to the 1-D FFT algorithm, two image partitioning strategies are considered first. The following two theorems regarding the two partitioning strategies are particularly interesting. Let $u = \frac{N}{2}$ and $v = \frac{r}{2}$ in the discussions to follow.

Theorem 5.1: Either $\mathbf{PE}_k$ and $\mathbf{PE}_{k+u}$ can access $p_{m,k}$ and $p_{m+v,k}$, for all $0 \leq k < u$ in parallel; or $\mathbf{PE}_k$ and $\mathbf{PE}_{k+u}$ can access $p_{m,k+u}$ and $p_{m+v,k+u}$, for all $0 \leq k < v$, where $m \in \mathbf{I}_N$ and $m_{\frac{n}{2}} = 0$.

Proof: Let integer A satisfy the condition $A = [01^{n-1}]$, and $l \in \mathbf{I}_N$. Then, $\mathbf{P}_A^l$ is parallel accessible by Theorem 3.2. Furthermore, let the routing vector of the Θ network be $0l$. It follows, by Theorem 3.5, that for all $k \in \mathbf{I}_N$, the kth processing element, $\mathbf{PE}_k$, is able to read pixel $p_{i,j}$, where

$$\begin{aligned} i &= \xi(Al \oplus \bar{A}k) \\ &= \xi([01^{n-1}]l \oplus [k_0 0^{n-1}]) \\ &= l_{\frac{n}{2}} \cdots l_{n-1} k_0 l_1 \cdots l_{\frac{n}{2}-1}, \end{aligned}$$

and

$$\begin{aligned} j &= \bar{A}l \oplus Ak \\ &= [l_0 0^{n-1}] \oplus [01^{n-1}]k \\ &= l_0 k_1 \cdots k_{n-1}. \end{aligned}$$

Now, if integer k satisfies the condition $k < u$. Then, $k_0 = 0$. It follows that $k = 0k_1 \cdots k_{n-1}$, and $k + u = 1k_1 \cdots k_{n-1}$. Let integer $m = l_{\frac{n}{2}} \cdots l_{n-1} 0 l_1 \cdots l_{\frac{n}{2}-1}$. Then, $m \in \mathbf{I}_N$ and $m_{\frac{n}{2}} = 0$. It follows that $m + v = l_{\frac{n}{2}} \cdots l_{n-1} 1 l_1 \cdots l_{\frac{n}{2}-1}$.

Now, consider the following two cases separately.

Case 1: Assuming $l_0 = 0$.

Integer j now assumes the value of $0k_1 \cdots k_{n-1}$. If k_0 is 0, then integer i assumes the value of $l_{\frac{n}{2}} \cdots l_{n-1} 0 l_1 \cdots l_{\frac{n}{2}-1}$, which is m. Otherwise, i assumes the value of $l_{\frac{n}{2}} \cdots l_{n-1} 1 l_1 \cdots l_{\frac{n}{2}-1}$ which is $m + v$. Consequently, PE_k and PE_{k+u} can read $p_{m,k}$ and $p_{m+v,k}$, respectively, for all $0 \leq k < u$ in parallel.

Case 2: Assuming $l_0 = 1$.

In this case, integer j assumes the value of $1k_1 \cdots k_{n-1}$ which is $k + u$. If k_0 is 0, then integer i assumes the value of m; otherwise the value of $m + v$, as in case 1. Consequently, PE_k and PE_{k+u} can read $p_{m,k+u}$ and $p_{m+v,k+u}$, respectively, for all $0 \leq k < u$ in parallel.

Q.E.D.

The interpretation of Theorem 5.1 is the following. In viewing the PE's as two independent one-dimensional subarrays of which processing elements 0 to $\frac{N}{2}-1$ form one subarray and processing elements $\frac{N}{2}$ to $N-1$ form another, the two subarrays can be applied in parallel to two rows of P which are $\frac{\sqrt{N}}{2}$ rows apart. Of particular interest, each processing element of a subarray can obtain two pixels which are at the same row and $\frac{N}{2}$ pixels apart with two read operations. Clearly, having such two rows placed on the two subarrays, each subarray is able to compute the T_1 vector for that row in constant time.

Theorem 5.2: Either PE_k and PE_{k+u} can access $p_{k,m}$ and $p_{k,m+v}$, respectively, for all $0 \leq k < u$ in parallel; or PE_k and PE_{k+u} can access $p_{k+u,m}$ and $p_{k+u,m+v}$, respectively, for all $0 \leq k < u$ in parallel, where $m \in I_N$ and $m_{\frac{n}{2}} = 0$.

Proof: Let integer A satisfy the condition $A = [0^{\frac{n}{2}}1^{\frac{n}{2}-1}]$, and integer $l \in I_N$. Then P_A^l is parallel accessible by Theorem 3.2. Furthermore, let the routing vector of the Θ network be $1\xi(l)$. It follows, by Theorem 3.5, that for all $k \in I_N$, the kth processing element, PE_k, is able to read pixel $p_{i,j}$, where

$$\begin{aligned} i &= \xi(Al \oplus \bar{A}\xi(k)) \\ &= [l_{\frac{n}{2}}0^{n-1}] \oplus [01^{n-1}]k \\ &= l_{\frac{n}{2}}k_1 \cdots k_{n-1}, \end{aligned}$$

and

$$\begin{aligned} j &= \bar{A}l \oplus A\xi(k) \\ &= [1^{\frac{n}{2}}01^{\frac{n}{2}-1}]l \oplus [0^{\frac{n}{2}}k_0 0^{\frac{n}{2}-1}] \\ &= l_0 \cdots l_{\frac{n}{2}-1}k_0 l_{\frac{n}{2}+1} \cdots l_{n-1}. \end{aligned}$$

Now, if k satisfies the condition $k < u$. Then, k_0 is 0. It follows that $k = 0k_1 \cdots k_{n-1}$, and $k + u = 1k_1 \cdots k_{n-1}$. Let integer $m = l_0 \cdots l_{\frac{n}{2}-1}0l_{\frac{n}{2}+1} \cdots l_{n-1}$. Then, $m \in I_N$ and $m_{\frac{n}{2}} = 0$. Clearly, $m + v = l_0 \cdots l_{\frac{n}{2}-1}1l_{\frac{n}{2}+1} \cdots l_{n-1}$.

Now, consider the following two cases.

Case 1: Assuming $l_{\frac{n}{2}} = 0$.

Integer i now assumes the value of $0k_1 \cdots k_{n-1}$. If k_0 is 0, then j equals to $l_0 \cdots l_{\frac{n}{2}-1}0l_{\frac{n}{2}+1} \cdots l_{n-1}$ which is m; otherwise, j assumes the value of $l_0 \cdots l_{\frac{n}{2}-1}1l_{\frac{n}{2}+1} \cdots l_{n-1}$ which is $m + v$. Thus, PE_k and PE_{k+u} can read $p_{k,m}$ and $p_{f,m+v}$, respectively, for all $k < u$ in parallel.

Case 2: Assuming $l_{\frac{n}{2}} = 1$.

In this case, integer i has the value of $1k_1 \cdots k_{n-1}$. If k_0 is 0, then j is m; otherwise it is $m + v$, the same as in case 1. Therefore, PE_k and PE_{k+u} can read

$p_{k+s,m}$ and $p_{k+s,m+v}$, respectively, for all $k < u$ in parallel.

Q.E.D.

Theorem 5.2 points out that the two linear subarrays can be applied in parallel to two columns of **P** which are $\frac{\sqrt{N}}{2}$ columns apart. In particular, each processing element can obtain two pixels which are on the same column and $\frac{N}{2}$ pixels apart. Obviously, having such two columns of **P** placed on the two subarrays, each subarray is able to compute the vector T_1 for that column in constant time. The use of these two theorems will become clear shortly.

5.2.1. A parallel algorithm for 1-D FFT

The parallel 1-D FFT algorithm is now presented as Algorithm **Double** which computes the 1-D FFT of two rows, or two columns of image **P** simultaneously.

Algorithm Double (A, l, α, d)

```
begin
1.  R0 ← M :(A, l, α);
2.  R1 ← M :(A, l+d, α);
3.  R2 := R0 + R1;  R1 := R0 − R1;
4.  for s := 1 to n − 1 do
      begin
5.      R0 := R1                    [φ^s 0 φ^(n−s−1)];
6.      R0 := R2                    [φ^s 1 φ^(n−s−1)];
7.      R0 ← R0 :(theta, 0^s 10^(n−s−1), 0);
8.      R1 := R0;  R0 := R2   [φ^s 0 φ^(n−s−1)];
9.      R1 := R1 × B_s;
10.     R2 := R0 + R1;  R1 := R0 − R1;
      end;
11. R2 → M :(A, l, α);
12. R1 → M :(A, l+d, α);
  end;
```

In the algorithm above, processing element instructions in company with a

mask will be executed by those processing elements whose indices match the mask. Instructions without the mask will be executed by all processing elements. In other words, a mask of the form $[\phi^n]$ does not appear explicitly in the algorithm in order to minimize confusion. In the algorithm, each processing element makes use of $n-1$ locations of its local memory to store the precomputed weights for the $n-1$ butterfly operations that it is going to perform. These local memory locations are represented as array **B**, where **B**, contains the weight involved in the butterfly operation in computing vector T_{s+1} from T_s as defined in the 1-D FFT formulation of Chapter 2.

Theorem 5.3: Let $A = [01^{n-1}]$, $l < u$, and $\alpha = 0$. **Double** (A, l, α, u) computes the 1-D FFT of row m and row $m + v$ of image **P** simultaneously, where $m = \xi(l)$.

Proof: Let vectors T_s^m and T_s^{m+v}, $1 \le s \le n$, be defined for row m and row $m+v$ of image **P**, respectively, as vectors T_s defined for vector x in the 1-D FFT formulation. Thus, it needs to be shown that at the termination of the algorithm, T_n^m and T_n^{m+v} will be stored in place of row m and row $m + v$ of image **P**, respectively.

By Theorem 5.1, after lines 1 and 2 have been executed, registers R_0 and R_1 of PE_f are loaded with $p_{m,f}$ and $p_{m,f+u}$ while registers R_0 and R_1 of PE_{f+u} are loaded with $p_{m+v,f}$ and $p_{m+v,f+u}$, for all $0 \le f < u$. Therefore, the execution of line 3 will load registers R_2 and R_1 of PE_f with $T_1^m(f)$ and $T_1^m(f+u)$, respectively. At the same time, it loads R_2 and R_1 of PE_{f+u} with $T_1^{m+v}(f)$ and $T_1^{m+v}(f+u)$, respectively.

The proof now proceeds by induction on s that at the end of the sth iteration of the loop at line 4, for all $(i-1)\frac{u}{2^s}) \le f < i\frac{u}{2^s}$, where $i \in \{1, \ldots, 2^s\}$, (1). Registers R_2 and R_1 of PE_f will contain $T_{s+1}^m(f + (i-1)\frac{u}{2^s})$ and $T_{s+1}^m(f + i\frac{u}{2^s})$, respectively; and (2). Registers R_2 and R_1 of PE_{f+u} will contain $T_{s+1}^{m+v}(f + (i-1)\frac{u}{2^s})$ and

$T^{m}_{s+1}(f + i\frac{u}{2^s})$, respectively. Thus, when $s = n-1$, for all $i-1 \leq f < i$, where $i \in \{1, \ldots, u\}$, PE_f will have $T^{m}_{n}(f+i-1)$ and $T^{m}_{n}(f+i)$ in its R_2 and R_1 registers, and PE_{f+u} will have $T^{m+v}_{n}(f+i-1)$ and $T^{m+v}_{n}(f+i)$ in its R_2 and R_1 registers. The subsequent write instructions at lines 11 and 12 will then store the outputs of the 1-D FFT of row m and row $m + v$ of image **P** in place of the inputs.

Thus it must be shown that the induction hypothesis is true. The following is to show that induction hypothesis (1) is true. The proof for induction hypothesis (2) is analogous, and thus omited.

Basis: $s = 1$.

Recall that R_2 and R_1 of PE_f, for all $0 \leq f < u$, contain $T^{m}_{1}(f)$ and $T^{m}_{1}(f+u)$ before the loop is entered. Consider the first iteration of the loop ($s = 1$). In line 5, PE_f, for all $0 \leq f < \frac{u}{2}$, moves $T^{m}_{1}(f+u)$ from R_1 to R_0 while in line 6 PE_f, for $\frac{u}{2} \leq f < u$, moves $T^{m}_{1}(f)$ from R_2 to R_0. The transfer instruction at line 7 exchanges the contents of R_0 of PE_f with that of $PE_{f+\frac{u}{2}}$ for all $0 \leq f < \frac{u}{2}$. After line 7 has been executed, register R_0 of PE_f contains $T^{m}_{1}(f+\frac{u}{2})$, and R_0 of $PE_{f+\frac{u}{2}}$ contains $T^{m}_{1}(f+u)$, for all $0 \leq f < \frac{u}{2}$. In line 8, PE_f, for all $0 \leq f < \frac{u}{2}$, loads its R_0 and R_1 registers with $T^{m}_{1}(f)$ and $T^{m}_{1}(f+\frac{u}{2})$, respectively. Note that R_0 and R_1 of PE_f, for all $\frac{u}{2} \leq f < u$, contain $T^{m}_{1}(f+\frac{u}{2})$ and $T^{m}_{1}(f+u)$, respectively. Therefore, after lines 9 and 10 have been executed, R_2 and R_1 of PE_f, for all $0 \leq f < \frac{u}{2}$, contain $T^{m}_{2}(f)$ and $T^{m}_{2}(f+\frac{u}{2})$, respectively, while R_2 and R_1 of PE_f, for all $\frac{u}{2} \leq f < u$, contain $T^{m}_{2}(f+\frac{u}{2})$ and $T^{m}_{2}(f+u)$, respectively.

Induction step: Assuming true for $s = k - 1$ and show true for $s = k$.

From the induction hypothesis when $s = k - 1$, registers R_2 and R_1 of PE_f contain $T_f^m(f + (i-1)\frac{u}{2^{k-1}})$ and $T_f^m(f + i\frac{u}{2^{k-1}})$, respectively, for all $(i-1)\frac{u}{2^{k-1}}) \leq f < i\frac{u}{2^{k-1}}$, where $i \in \{1, ..., 2^{k-1}\}$. Now, consider the kth iteration of the for loop. In line 5, PE_f, for all $(i-1)\frac{u}{2^{k-1}} \leq f < (i-1)\frac{u}{2^{k-1}}+\frac{u}{2^k}$, moves $T_f^m(f+i\frac{u}{2^{k-1}})$ from its R_1 to R_0. Correspondingly, in line 6, PE_f, for all $(i-1)\frac{u}{2^{k-1}}+\frac{u}{2^k} \leq f < i\frac{u}{2^{k-1}}$, moves $T_f^m(f + (i-1)\frac{u}{2^{k-1}})$ from R_1 to R_0. The transfer instruction at line 7 exchanges the contents of R_0 of PE_f with that of $PE_{f+\frac{u}{2^k}}$, for all $(i-1)\frac{u}{2^{k-1}} \leq f < (i-1)\frac{u}{2^{k-1}}+\frac{u}{2^k}$. Thus, after line 7 has been executed, R_0 of PE_f contains $T_f^m(f + (i-1)\frac{u}{2^{k-1}}+\frac{u}{2^k})$ that is $T_f^m(f + (2i-1)\frac{u}{2^k})$ while R_0 of $PE_{f+\frac{u}{2^k}}$ contains $T_f^m(f + i\frac{u}{2^{k-1}})$. In line 8, PE_f, for all $(i-1)\frac{u}{2^{k-1}} \leq f < (i-1)\frac{u}{2^{k-1}}+\frac{u}{2^k}$, moves $T_f^m(f + (2i-1)\frac{u}{2^k})$ from R_0 to R_1, and $T_f^m(f + (i-1)\frac{u}{2^{k-1}})$, i.e., $T_f^m(f + (2i-2)\frac{u}{2^k})$ from its R_1 to R_0. Note that at this point in each PE_f, for all $(i-1)\frac{u}{2^{k-1}}+\frac{u}{2^k} \leq f < i\frac{u}{2^{k-1}}$, R_0 contains $T_f^m(f + i\frac{u}{2^{k-1}}-\frac{u}{2^k})$, i.e., $T_f^m(f + (2i-1)\frac{u}{2^k})$, and R_1 contains $T_f^m(f + i\frac{u}{2^{k-1}})$, i.e., $T_f^m(f + 2i\frac{u}{2^k})$. Consequently, the execution of lines 9 and 10 will load R_2 and R_1 of PE_f with $T_f^m(f + (2i-2)\frac{u}{2^k})$ and $T_f^m(f + (2i-1)\frac{u}{2^k})$, respectively, for all $(2i-2)\frac{u}{2^k} \leq f < (2i-1)\frac{u}{2^k}$. At the same time, R_2 and R_1 of PE_f are loaded with $T_f^m(f + (2i-1)\frac{u}{2^k})$ and $T_f^m(f + 2i\frac{u}{2^k})$, respectively, for all $(2i-1)\frac{u}{2^k} \leq f < 2i\frac{u}{2^k}$. Now, let $i \in \{1, ..., 2^k\}$. Then, each i can be expressed as either $2i$ or $2i-1$

for some $i \in \{1, ..., 2^{k-1}\}$. It is then easy to show that registers R_2 and R_1 of PE_f contain $T^m_{k+1}(f+(i-1)\frac{u}{2^k})$ and $T^m_{k+1}(f+i\frac{u}{2^k})$, respectively, for all $-(i-1)\frac{u}{2^k}) \le f < i\frac{u}{2^k}$, where $i \in \{1, ..., 2^k\}$. This completes the proof.

Q.E.D.

Measuring the time complexity of the algorithm in terms of the number of instructions to be executed, the following theorem results.

Theorem 5.4: Algorithm **Double** is of time complexity $O(\log N)$.

Proof: It is straightforward to analyze the cost of algorithm **Double**. There are 2 read instructions in lines 1 and 2, two additions in line 3 before the loop is entered. The loop at line 4 iterates $n-1$ times. At each iteration of the loop, it takes four register instructions in lines 4, 5, and 8; one transfer instruction in line 7; one multiplication in line 9; and two additions in line 10. The theorem follows, since $n = \log N$.

Q.E.D.

As an immediate consequence of Theorems 5.2 and 5.4, Theorem 5.5 follows.

Theorem 5.5: Let $A = [0^{\frac{u}{2}}1 0^{\frac{u}{2}-1}]$, $l < u$, and $\alpha = 1$. **Double** $(A, \xi(l), \alpha, v)$ computes the 1-D FFT of column m and column $m+v$ of image **P** simultaneously, where $m = \xi(l)$, in $O(\log N)$ time.

The well known 1-D FFT algorithm requires $O(N \log N)$ time steps to compute the 1-D DFT of a vector of N elements in the uniprocessor environment. In contrast, algorithm **Double** takes $O(\log N)$ time steps to compute the 1-D FFT of two rows or two columns of **P** in parallel. Moreover, Theorems 5.3 and 5.5 claim that algorithm **Double** can be applied to two rows and two columns of **P** equally well.

The remainder of this section is devoted to the illustration of the three major steps of Algorithm **Double** in computing the 1-D FFT of two rows of image **P** for the case of $N = 16$ ($u = 8$ and $v = 2$). As $N = 16$, A and α will assume respectively the values of $[01^3]$ and 0 according to Theorem 5.3. When $l = 0$, **Double**(A, l, α, u) computes the 1-D FFT of row 0 and row 2 in place simultaneously as follows.

In referring to Algorithm **Double**, the first line will place the pixel $p_{0,k}$ in $\mathbf{PE}_k$ and pixel $p_{2,k}$ in $\mathbf{PE}_{k+8}$, for all $0 \leq k \leq 7$, in parallel. This is illustrated in Fig. 5.1 by the access matrix which depicts the placement of the two left halves of row 0 and row 2 on the two subarrays of processing elements.

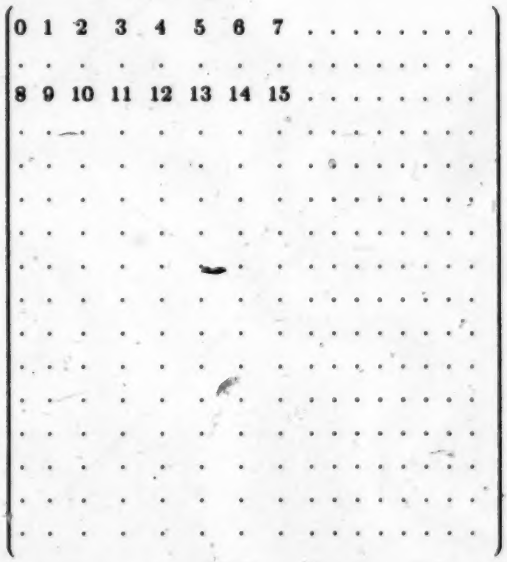

Fig. 5.1. Accessing the two left halves of row 0 and row 2.

Then, the second line will place the pixel $p_{0,k+8}$ in $\mathbf{PE}_k$ and pixel $p_{2,k+8}$ in $\mathbf{PE}_{k+8}$, for all $0 \leq k \leq 7$, in parallel. The placement of the two right halves of row 0 and row 2 on the two subarrays is illustrated by the access matrix in Fig. 5.2.

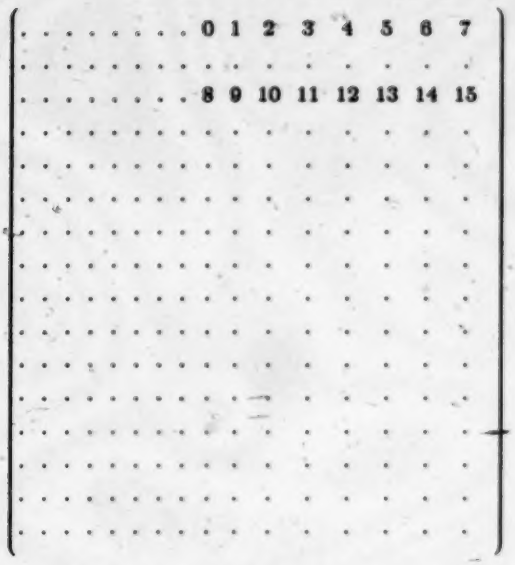

Fig. 5.2 Accessing the two right halves of row 0 and row 2.

As can be seen from these two access matrices, PE_k is now provided with two pixels of row 0, i.e., $p_{0,k}$ and $p_{0,k+8}$, for all $0 \leq k \leq 7$. At the same time PE_{k+8} is provided with two pixels of row 2, i.e., $p_{2,k}$ and $p_{2,k+8}$, for all $0 \leq k \leq 7$. The two pixels in PE_k are 8 pixels apart. Thus, at line 3, PE_k computes the two elements of vector T_1^0, i.e., $T_1^0(k)$ and $T_1^0(k+8)$. Similarly, PE_{k+8} computes the two elements of T_1^2, i.e., $T_1^2(k)$ and $T_1^2(k+8)$.

The for loop at line 4 is to be executed three times in order to compute T_4^0 and T_4^2. During the 1st iteration of the loop, the elements of T_1^0 and T_1^2 are moved among the PE's at line 7 according to the following mapping of PE's to themselves:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 & 12 & 13 & 14 & 15 & 8 & 9 & 10 & 11 \end{pmatrix}$$

where a mapping π is specified by placing $\pi(k)$ below k in the 2×16 array to mean that PE_k is connected to $PE_{\pi(k)}$ through the Θ network.

A quick examination of this mapping will reveal the fact that each processing element will be provided with two elemets of either T_1^0 or T_1^2, which are 4 elements apart, after the execution of line 7. The rest of the loop computes T_2^0 and T_2^2 in parallel.

The second iteration of the loop is to compute T_3^0 and T_3^2. The elements of T_2^0 and T_2^2 are first permuted among the PE's through the Θ network according to the following mapping:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 2 & 3 & 0 & 1 & 6 & 7 & 4 & 5 & 10 & 11 & 8 & 9 & 14 & 15 & 12 & 13 \end{pmatrix}.$$

This mapping will provide each processing element with two elements of either T_2^0 or T_2^2, which are 2 elements apart. T_3^0 and T_3^2 are then computed in parallel.

Finally, the last iteration of the loop will compute T_4^0 and T_4^2. The elements of T_3^0 and T_3^2 are first permuted among the PE's according to the following mapping:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 1 & 0 & 3 & 2 & 5 & 4 & 7 & 6 & 9 & 8 & 11 & 10 & 13 & 12 & 15 & 14 \end{pmatrix}.$$

This mapping allows each processing element to have two elements of either T_3^0 or T_3^2, which are 1 element apart. The processing elements then compute T_4^0 and T_4^2 in parallel.

Upon termination of the loop, at line 11, the two left halves of vector T_4^0

and vector T_4^1 are written back to **M** in place of the two left halves of row 0 and row 2. Similarly, at line 12, the two right halves of vector T_4^0 and vector T_4^1 are written back to **M** in place of the two right halves of row 0 and row 2.

5.2.2. A parallel algorithm for 2-D FFT

It has been shown by Theorem 5.3 that algorithm **Double** presents an in-place computation of 1-D FFT of two rows of image **P** simultaneously. Therefore, applying algorithm **Double** u times to image **P** rowwise will replace each row of image **P** by the 1-D DFT of that row, thereby generating an intermediate array. However, algorithm **Double** can also be used to compute the 1-D DFT of two columns of image **P** in place simultaneously according to Theorem 5.5. Thus, applying algorithm **Double** u times to this intermediate array columnwise will replace each column of the intermediate array by the 1-D DFT of that column, resulting a new array. This process corresponds to the computation of 2-D DFT of image **P** by decomposing it into row and column 1-D DFT's. The newly obtained array is, however, not the 2-D DFT of image **P** as yet, since some of its rows and columns are not in the right order according to the 1-D FFT formulation previously described. Action must be then taken to bring those rows as well as columns into the correct order. It can be shown that there are $N - v$ such rows and $N - v$ such columns in the new array. Reordering the new array can be done in $O(N)$ time if the new image is partitioned into rows and columns, and exchange row i with row i' only if $i \neq i'$ for all $i \in I_N$. The same process is then repeated for the resulting image columnwise. The parallel 2-D FFT algorithm is now presented below as algorithm **E**.

Algorithm E

begin

1. $A := 01^{n-1}$;
2. $\alpha := 0$;
3. for $l := 0$ to $u - 1$ do
4. **Double** (A, l, α, u);
5. $A := 0^{\frac{n}{2}} 10^{\frac{n}{2}-1}$;
6. $\alpha := 1$;
7. for $l := 0$ to $u - 1$ do
8. **Double** $(A, \xi(l), \alpha, v)$;
9. reordering the new image;

end;

Using Theorems 5.3, 5.4, and 5.5, it is possible to show the following.

Theorem 5.6: Algorithm **E** computes the 2-D FFT of image **P** in place in time $O(N \log N)$.

Compared to the well known 2-D FFT algorithm which needs $O(N^2 \log N)$ time steps to compute the 2-D DFT of image **P** in the uniprocessor environment, the parallel algorithm can do it N times as fast on the proposed architecture. Such a speed-up factor is achieved not only by the two partitions used but also by the powerful permutation function of the theta network.

5.3. Matrix multiplication

This section considers multiplication of two $N \times N$ matrices on the proposed multiple-processor. Let $\mathbf{X} = [x_{i,k}]$ and $\mathbf{Y} = [y_{k,j}]$ be $N \times N$ matrices. The multiplication of $\mathbf{X}$ and $\mathbf{Y}$ generates a product matrix $\mathbf{Z} = \mathbf{X} \times \mathbf{Y} = [z_{i,j}]$ of dimension $N \times N$. The elements of $\mathbf{Z}$ are related to the elements of $\mathbf{X}$ and $\mathbf{Y}$ by:

$$z_{i,j} = \sum_{k=0}^{N-1} x_{i,k} \times y_{k,j}$$

where $i,j \in \mathbf{I}_N$. There are N^3 multiplications and N^3 additions to be performed in

the above formulation.

5.3.1. A sequential matrix multiplication algorithm

The conventional sequential algorithm based on the above formulation for the multiplication of two $N \times N$ matrices requires $O(N^3)$ multiplications and additions in the uniprocessor environment. Systolic algorithms based on the same formulation for matrix multiplication achieve the time bound of $O(N)$ employing $O(N^2)$ processors.

The dominating cost of matrix multiplication is the number of multiplications to be performed. Especially, when the processor is very simple by itself, and this is overwhelmingly the case in the SIMD computing environment where computation power is accomplished by massive parallelism. In the following, a sequential algorithm for matrix multiplication [48] is first described, which halves the number of multiplications required by conventional algorithms, in preparation for the development of a parallel version of the algorithm on the proposed architecture. The sequential algorithm consists of the following three steps.

Step 1. Compute, for $i \in \mathrm{I}_N$,

$$H_i = \sum_{k=0}^{\frac{N}{2}-1} x_{i,2k} \times x_{i,2k+1} ;$$

Step 2. Compute, for $j \in \mathrm{I}_N$,

$$V_j = \sum_{k=0}^{\frac{N}{2}-1} y_{2k,j} \times y_{2k+1,j} ;$$

Step 3. Compute, for $i,j \in \mathrm{I}_N$,

$$z_{i,j} = -H_i - V_j + \sum_{k=0}^{\frac{N}{2}-1} (x_{i,2k}+y_{2k+1,j}) \times (x_{i,2k+1}+y_{2k,j}) .$$

In the above formulation of the matrix multiplication, steps 1 and 2 take N^2 multiplications and N^2-2N additions. Step 3 takes $\frac{N^2}{2}$ multiplications and

$N^2(\frac{3N}{2}+1)$ additions. The total cost of the algorithm is, therefore, $\frac{N^3}{2}+N^2$ multiplications and $N^3+\frac{3N^2}{2}-2N$ additions. The fact that this algorithm takes only half of the number of multiplications required by conventional method makes it a faster method when the multiplication of two matrices is performed on any computer where multiplication operation is much costly than addition operation.

5.3.2. A parallel matrix multiplication algorithm

An efficient parallel implementation of the fast matrix multiplication method on the proposed architecture will be presented in this section. It is assumed that there are two parallel memory systems in the system denoted by $\mathbf{M}^0$ and $\mathbf{M}^1$, respectively. With very little additional control hardware added to the multiple-processor, the N processing elements can access any of these two parallel memory systems at a time. Assume matrices $\mathbf{X}$ and $\mathbf{Y}$ are stored in $\mathbf{M}^0$ and $\mathbf{M}^1$, respectively. And the resulting product matrix $\mathbf{Z}$ will be stored in place of the matrix $\mathbf{X}$.

The parallel algorithm to follow makes use of two N-partitions of $\mathbf{X}$ and $\mathbf{Y}$ in combination to achieve its computational efficiency. According to Corollary 3.8, the N processing elements can access any column of matrix $\mathbf{X}$ at a time in parallel. This architectural feature makes it possible for $\mathbf{PE}_i$ to compute H_i, for all $i \in \mathbf{I}_N$, in parallel. On the other hand, by Corollary 3.7, the N processing elements can access any row of matrix $\mathbf{Y}$ at a time. Thus, it is possible for $\mathbf{PE}_j$ to compute V_j, for all $j \in \mathbf{I}_N$, in parallel.

The parallel algorithm is now formally presented below as Algorithm **F**, where lines 1 to 5, lines 6 to 10, and lines 11 to 24 correspond to steps 1, 2, and 3 of the fast method, respectively.

Algorithm F

```
begin
1.  H := 0; A := 0^n; α := 1;
2.  for i := 0 to N/2 − 1 do
      begin
3.      R0 ← M^0 :(A, 2i, α);
4.      R1 ← M^0 :(A, 2i+1, α);
5.      H += R0 × R1;
      end;
6.  V := 0; A := 1^n; α := 0;
7.  for j := 0 to N/2 − 1 do
      begin
8.      R0 ← M^1 :(A, ξ(2j), α);
9.      R1 ← M^1 :(A, ξ(2j+1), α);
10.     V += R0 × R1;
      end;
11. for i := 0 to N−1 do
      begin
12.     σ := 0;
13.     R2 ← M^0 :(A, ξ(i), α);
14.     for j := 0 to N/2 − 1 do
          begin
15.           R0 ← M^1 :(A, ξ(2j), α);
16.           R1 ← M^1 :(A, ξ(2j+1), α);
17.           R2 → γ :CUN     [2j];
18.           R1 += γ;
19.           R2 → γ :CUN     [2j+1];
20.           R0 += γ;
21.           σ += R0 × R1;
          end;
22.     H → γ :CUN     [i];
23.     R0 := σ − H − γ;
24.     R0 → M^0 :(A, ξ(i), α);
      end;
end;
```

In Algorithm **F**, each processing element uses three locations of its local memory which are represented as variables H, V, and σ, respectively. On termination of the loop at line 2, H of $\mathbf{PE}_i$ will contain the value of H_i for all $i \in \mathbf{I}_N$. Upon termination of the loop at line 7, V of $\mathbf{PE}_i$ will contain the value of V_i for all $i \in \mathbf{I}_N$. With H_i and V_i stored in $\mathbf{PE}_i$ for all $i \in \mathbf{I}_N$, the N processing elements compute one row of matrix **Z** at each iteration of the loop at line 11. Instruc-

tion of the form "$R_2 \rightarrow \gamma$:CUN [mask]" in the algorithm is a CUN instruction that loads its γ register with the contents of R_2 of the processing element whose index matches the mask field of the instruction. The correctness of the algorithm is given below.

Theorem 5.7: Algorithm **F** computes the product of two matrices of dimension $N \times N$.

Proof: It will be shown that the for loops at lines 2, 7, and 11 correctly implement steps 1, 2, and 3 of the fast matrix multiplication method, respectively.

The correctness of the loop at line 2 will be proved by induction on i that at the end of the ith iteration of the loop variable H of PE_f will contain the value of $\sum_{k=0}^{i} x_{f,2k} \times x_{f,2k+1}$, for all $f \in I_N$. Thus, when $i = \frac{N}{2}$, H of PE_f will contain the value of H_f.

Basis: $i = 0$.

Consider the first iteration of the loop. After the execution of lines 3 and 4, R_0 and R_1 of PE_f are loaded with $x_{f,0}$ and $x_{f,1}$, respectively, by Corollary 3.8. After line 5 is executed, H of PE_f contains clearly $x_{f,0} \times x_{f,1}$.

Induction step: Assuming true for $i = j-1$ and show true for $i = j$.

Now, consider the jth iteration of the loop. After the execution of lines 3 and 4, R_0 and R_1 of PE_f are loaded with $x_{f,2j}$ and $x_{f,2j+1}$, respectively, by Corollary 3.8. Invoking the inductive hypothesis, after line 5 is executed, H of PE_f contains the value of $\sum_{k=0}^{j-1} x_{f,2k} \times x_{f,2k+1} + x_{f,2j} \times x_{f,2j+1}$, for all $f \in I_N$. The induction step then follows.

To prove that at the termination of the loop at line 7, the variable V of

PE_f will contain the value of V_f for all $f \in I_N$ is analogous, and omited here.

In order to prove that the loop at line 11 is a functionally correct implementation of step 3 of the fast method needs to show that at the end of the ith iteration of the loop, the ith row of the matrix $\mathbf{X}$ will be replaced by the ith row of the product matrix $\mathbf{Z}$.

Now, consider the ith iteration of the loop. First, every processing element clears its own variable σ at line 12, and the execution of line 13 loads register R_2 of PE_f with $x_{i,f}$ according to Corollary 3.7. The proof now proceeds by induction on j that at the end of the jth iteration of the loop at line 14, variable σ of PE_f will contain the value of $\sum_{k=0}^{j}(x_{i,2k}+y_{2k+1,f})(x_{i,2k+1}+y_{2k,f})$, for all $f \in I_N$. Thus, when $j = \frac{N}{2}-1$, σ of PE_f will contain the value of $\sum_{k=0}^{\frac{N}{2}-1}(x_{i,2k}+y_{2k+1,f})(x_{i,2k+1}+y_{2k,f})$. Subsequently, in line 22, the CUN loads its own γ register with H_i, i.e., the value of H of PE_i. Thus, line 23 loads register R_0 of PE_f with $x_{i,f}$, for all $f \in I_N$. Therefore, $x_{i,f}$ is written in place of $x_{i,f}$ by the write instruction at line 24, according to Corollary 3.7. Thus, it must be shown that the induction hypothesis is true.

Basis: $j = 0$.

At the first iteration of the loop, after lines 15 and 16 have been executed, R_0 and R_1 of PE_f are loaded with $y_{0,f}$ and $y_{1,f}$, respectively. In line 17, The CUN only reads $x_{i,0}$, i.e., the content of R_2 of PE_0, into its γ register. Each processing element adds $x_{i,0}$, which is broadcast by the CUN, to its R_1 register in line 18. Thus, line 18 arguments the contents of R_1 of PE_f to be $x_{i,0}+y_{1,f}$. Similarly, R_0 of PE_f will contain $x_{i,1}+y_{0,f}$ after line 20 is executed. Consequently, at the end of the first iteration σ of PE_f contains $(x_{i,0}+y_{1,f})(x_{i,1}+y_{0,f})$.

Induction step: Assume true for $j = m-1$, show true for $j = m$.

Consider the mth iteration of the loop. $\mathbf{R}_0$ and $\mathbf{R}_1$ of $\mathbf{PE}_j$ are loaded with $y_{2m,j}$ and $y_{2m+1,j}$ in lines 15 and 16, respectively. After the execution of lines 17 and 18, $\mathbf{R}_1$ and $\mathbf{R}_0$ of $\mathbf{PE}_j$ contain $x_{i,2m} + y_{2m+1,j}$ and $x_{i,2m+1} + y_{2m,j}$, respectively. Then, in line 21, $\mathbf{PE}_j$ updates its variable σ to be

$$\left(\sum_{k=0}^{m-1}(x_{i,2k}+y_{2k+1,j})(x_{i,2k+1}+y_{2k,j})\right) + (x_{i,2m}+y_{2m+1,j})(x_{i,2m+1}+y_{2m,j}),$$

where the first term is from the induction hypothesis. The induction hypothesis, therefore, follows.

This completes the proof.

Q.E.D.

Theorem 5.8: Algorithm **F** is of time complexity $O(N^2)$.

Proof: The loop at line 2 is of $O(N)$ time complexity in terms of the number of instructions to be performed. So is the loop at line 7. Similarly, the loop at line 11 takes $O(N^2)$ instructions.

Q.E.D.

5.4. Conclusion

Two parallel algorithms in the category of global transforms have been implemented on the proposed architecture. The 2-D FFT algorithm achieves its computational efficiency by the use of two suitable N-partitions of **P** in conjunction with a set of permutations realizable by the theta network. As can be seen, although the computational structure of the 2-D FFT is different from neighborhood operation in nature, the architecture is still able to support its parallel implementation efficiently.

The parallel matrix multiplication algorithm takes the advantage of the sequential algorithm in reducing the number of multiplications. It is the two N-

partitions of $\mathbf{P}$ based on Corollaries 3.7 and 3.8 that help the parallel implementation of the sequential algorithm. It should be pointed out that matrix multiplication can be useful operation for image processing if it can be done in a reasonable speed, since transformations of images can be defined in terms of matrix multiplications.

CHAPTER 6

CONCLUSION

This thesis has taken the approach to embody a set of useful image partitioning strategies in computer architecture to achieve its computational efficiency for a wider range of image processing tasks than previous image processors. This approach is the result of the view of a parallel image processor as a hardware implementation of the partition strategy chosen for the class of image processing tasks of interest. The storage scheme together with the theta interconnection network introduced in Chapter 3 play an essential role in this regard. First, the storage scheme makes it possible that any part of the image under a given partition is parallel accessible. Second, the theta network is capable of moving every pixel of a given part to the desired processing element in one pass for subsequent processing.

The algorithm-architecture matching is accomplished by choosing one or more pertinent partition strategies in combination to couple the raw computational power of the multiple-processor to the intended image processing task. The algorithm-architecture matching can be done dynamically according to the partition strategies chosen for the image processing tasks to be performed. This point has been demonstrated in Chapters 4 and 5 in detail, where image processing algorithms with diverse communication structures can be executed on the proposed architecture with a high degree of parallelism. In designing a parallel algorithm on the proposed architecture the first step is to decide which partition strategies to use in order to match the problem at hand. This requires not only a careful study of the computational structure of the problem but also the properties of the partition strategies in the machine. It may not be necessary that a

pertinent partition strategy exists that offers the algorithm-architecture matching for that particular problem. After all, no claim has been made insofar that the proposed architecture is a general-purpose parallel computer. What can be claimed is that the proposed computer architecture can match at least three classes of image processing algorithms, namely, local neighborhood operations, global neighborhood operations, and global transforms.

Several interesting problems which could be the topic of further studies became apparent during the course of this research. The use of the partition strategies for image processing tasks outside the three classes can be researched further. One of the most important, yet least understood, aspects of parallel processing is the effect of interconnections of memories and processing elements on computation speed. Several interconnection networks for processor-memory and interprocessor communication have been proposed for parallel processing in the literature [3, 6, 16, 23, 29, 31, 41]. The proposed theta network is capable of realizing many useful permutations which can not be realized by the existing ones in a single pass with simple control [24, 39, 46]. Although it was proposed for parallel image processing, the richer interconnections of this network may make it a promising interconnection pattern for other applications as well. Further exploration of its permutation capability is of both practical and theoretical interest. This will allow one to discover other useful permutations to support certain parallel image processing algorithms. It might also provide insights into the feasibilities of the architecture for applications other than image processing.

The versatility of the proposed architecture promises to make it an extremely exciting new tool for various types of image processing and pattern recognition applications. It is an interesting topic of further work to prototype the proposed architecture, and to look into the effects that this versatile architecture will have on parallel image processing.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

2. G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746-757, Aug. 1968.

3. K.E. Batcher, "Sorting network and its applications," *Proc. AFIPS-SJCC*, vol. 32, pp. 307-314, 1968.

4. K.E. Batcher, "STARAN parallel processor system," *Proc. Int'l Conf. Parallel Processing*, pp. 65-71, Aug. 1976.

5. K.E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, vol. C-29, pp. 836-840, Sept. 1980.

6. L.N. Bhuyan, "An analysis of processor-memory interconnection networks," *IEEE Trans. Comput.*, vol. C-34, pp. 297-283, Mar. 1985.

7. P.J. Burt and E.H. Adelson, "The Laplacian pyramid as a compact image code," *IEEE Trans. Comm.*, vol. C-31, pp. 532-540, April 1983.

8. L.P. Cordella, M.J.B. Duff, and S. Levialdi, "An analysis of the computational cost in image processing," *IEEE Trans. Comput.*, vol. C-27, pp. 904-910, Dec. 1978.

9. P.E. Danielsson and S. Levialdi, "Computer architectures for pictorial information systems," *Computer*, vol. 14, pp. 53-67, Nov. 1981.

10. M.J.B. Duff, "CLIP 4- A large scale integrated circuit array parallel processor," *Proc. 3rd Int'l Conf. Pattern Recognition*, pp. 728-733, 1976.

11. M.J.B. Duff and T.J. Fountain, ed, *Cellular Logic Image Processing*, Academic Press, 1986.

12. C.R. Dyer, "Pyramid algorithms and machines," in *Multi-computers and Image Processing*, ed. L. Uhr, pp. 409-420, Academic Press, 1982.

13. T. Ericsson and P.E. Danielsson, "LIPP- A SIMD multiprocessor architecture for image processing," *Proc. 10th Annu. Symp. Comput. Arch.*, pp. 395-400, 1983.

14. O. Ersoy, "Semisystolic array implementation of circular skew circular, and linear convolution," *IEEE Trans. Comput.*, vol. C-34, pp. 190-196, Feb. 1985.

15. A.L. Fasher and H.T. Kung, "Synchronization large VLSI processor arrays," *IEEE Trans. Comput.*, vol. C-34, Aug. 1985.

16. T.Y. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Comput.*, vol. C-23, pp. 309-318, Mar. 1974.

17. M.J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948-960, Sept. 1972.

18. C. Guerra, "Systolic algorithms for local operations on images," *IEEE Trans. Comput.*, vol. C-35, pp. 73-76, Jan. 1986.

19. W.D. Hillis, *The connection machine*, MIT Press, 1985.

20. L.H. Jamieson, P.T. Tueller, and H.J. Siegel, "FFT algorithms for SIMD parallel processing systems," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 48-71, 1986.

21. H.T. Kung, "Why systolic architectures?," *Computer*, vol. 15, pp. 37-46, Jan. 1982.

22. H.T. Kung and J.A. Webb, "Mapping image processing operations onto a linear systolic machine," *Distributed Computing*, vol. 1, pp. 246-257, Oct. 1986.

23. D. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145-1155, Dec. 1975.

24. J. Lenfant, "A versatile mechanism to move data in an array processor," *IEEE Trans. Comput.*, vol. C-34, June 1985.

25. G.J. Li and B.W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 66-77, Jan. 1985.

26. R.M. Lougheed, D.L. McCubbrey, and S.R. Sternberg, "Cytocomputers: architectures for image processing," *Proc. Workshop on Picture Data Descr. and Management*, pp. 281-286, Pacific Grove, Calif., Aug. 1980.

27. K. Luetjen, P. Gemma, and H. Ischen, "Flip: a flexible multiprocessor system for image processing," *Proc. 5th Int'l Conf. Pattern Recognition*, vol. 1, pp. 326-328, Miami, Dec. 1980.

28. E.Q. Nwachukwu, "Address generation in an array processor," *IEEE Trans. Comput.*, vol. C-34, pp. 170-173, Feb. 1985.

29. J.H. Patel, "Performance of processor-memory interconnections for multiprocessors," *IEEE Trans. Comput.*, vol. C-30, pp. 771-780, Oct. 1983.

30. M.C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *JACM*, vol. 15, pp. 252-264, Apr. 1968.

31. M.C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458-473, May 1977.

32. K. Preston, M.J.B. Duff, and S. Levialdi, "Basis of cellular logic with some applications in medical image processing," *Proc. IEEE*, vol. 67, pp. 826-856, May 1979.

33. S.F. Reddaway, "DAP- A distributed array processor," *Proc. 1st Annu. Symp. Comput. Arch.*, pp. 61-65, Florida, Dec. 1973 .

34. A.P. Reeves, "A systematically designed binary array processor," *IEEE Trans. Comput.*, vol. C-29, pp. 278-287, March 1980.

35. A.P. Reeves, "Parallel computer architectures for image processing," *Computer vision, Graphics, and Image Processing*, vol. 25, pp. 68-88, Feb. 1984.

36. A.P. Reeves and J.D. Bruner, "Efficient function implementation for a bit-serial parallel processor," *IEEE Trans. Comput.*, vol. C-29, pp. 841-844, Sept. 1980.

37. A. Rosenfeld, "Parallel image processing using cellular arrays," *Computer*, vol. 3, pp. 14-20, Jan. 1983.

38. A. Rosenfeld and A.C. Kak, *Digital picture processing*, Academic Press, 1976.

39. H.J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Comput.*, vol. C-26, pp. 153-161, Feb. 1977.

40. H.J. Siegel, "PASM- A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934-947, Dec. 1981.

41. H.S. Stone, "Parallel processing with perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153-161, Feb. 1971.

42. H.S. Stone, ed., *Introduction to computer architecture*, SRA, 1980.

43. S.L. Tanimoto, "Algorithms for median filtering of images on a pyramid machine," in *Computing Structures for Image processing*, ed. M.J.B. Duff, pp. 71-74, Academic Press, 1983.

44. S.H. Unger, "A computer oriented toward spatial problems," *Proc. IRE*, vol. 46, pp. 1744-1750, Oct. 1958.

45. S.H. Unger, "Pattern detection and recognition," *Proc. IRE*, pp. 1737-1752, 1959.

46. A. Varma and C.S. Raghavendra, "On permutations passable by the Gamma network," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 72-91, 1986.

47. D.C. Van Voorhis and T.H. Morrin, "Memroy systems for image processing," *IEEE Trans. Comput.*, vol. C-27, pp. 113-125, Feb. 1978.

48. V. Zakharov, "Parallelism and array processing," *IEEE Trans. Comput.*, vol. C-33, pp. 45-77, Jan. 1984.